

WL-TR-94-1015



Application of Genetic Algorithms to
Function Decomposition in Pattern Theory

Michael J. Noviskey
Timothy D. Ross &
David A. Gadd

Applications Branch
Mission Avionics Division

Mark Axtell

Veda Inc., Dayton OH

Jan 1994

Interim Report for Period Jan 1993 to Aug 1993

Approved for public release; distribution limited

AVIONICS DIRECTORATE
WRIGHT LABORATORY
AIR FORCE MATERIEL COMMAND
WRIGHT-PATTERSON AIR FORCE BASE, OHIO 45433-7623

DTIC QUALITY INSPECTED 3

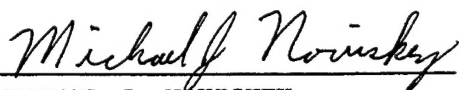
19970804 050

NOTICE

When Government drawings, specifications, or other data are used for any purpose other than in connection with a definitely Government-related procurement, the United States Government incurs no responsibility or any obligation whatsoever. The fact that the government may have formulated or in any way supplied the said drawings, specifications, or other data, is not to be regarded by implication, or otherwise in any manner construed, as licensing the holder, or any other person or corporation; or as conveying any rights or permission to manufacture, use, or sell any patented invention that may in any way be related thereto.

This report is releasable to the National Technical Information Service (NTIS). At NTIS, it will be available to the general public, including foreign nations.

This technical report has been reviewed and is approved for publication.



MICHAEL J. NOVISKEY
Project Engineer
Surface Strike Section
Targeting Attack Systems Branch



EDWARD L. HAMILTON, Chief
Surface Strike Section
Targeting Attack Systems Branch
Mission Avionics Division



WILLIAM E. MOORE, Chief
Targeting Attack Systems Branch
Mission Avionics Division
Avionics Directorate

If your address has changed, if you wish to be removed from our mailing list, or if the addressee is no longer employed by your organization please notify WL/AART-2, WPAFB, OH 45433-7408 to help us maintain a current mailing list.

Copies of this report should not be returned unless return is required by security considerations, contractual obligations, or notice on a specific document.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.

1. AGENCY USE ONLY (Leave blank)		2. REPORT DATE 26 01 94	3. REPORT TYPE AND DATES COVERED Interim Jan 93 - Aug 93
4. TITLE AND SUBTITLE Application of Genetic Algorithms to Functional Decomposition in Pattern Theory		5. FUNDING NUMBERS WU 0100AA13 PE 62204F PR 0100 TA AA WU 13	
6. AUTHOR(S) Michael J. Noviskey, Timothy D. Ross, David A. Gadd, and Mark Axtell		8. PERFORMING ORGANIZATION REPORT NUMBER	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Avionics Directorate, WL, AFMC WL/AART-2 Bldg 22 2690 C Street Suite 1 Wright-Patterson AFB OH 45433-7408		Veda Inc. Dayton OH 45431	
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Avionics Directorate Wright Laboratory Air Force Materiel Command Wright-Patterson AFB OH 45433-7623 POC: Michael Noviskey, WL/AART-2, WPAFB OH; 937-255-3215		10. SPONSORING/MONITORING AGENCY REPORT NUMBER WL-TR-94-1015	
11. SUPPLEMENTARY NOTES			
12a. DISTRIBUTION/AVAILABILITY STATEMENT Approved for public release; distribution unlimited		12b. DISTRIBUTION CODE	
13. ABSTRACT (Maximum 200 words) This report documents use of genetic algorithms for finding partitions which lead to optimal decomposition of boolean functions in the Ashenhurst - Curtis method of functional decomposition. This problem apparently grows exponentially as the number of input variables increase, but is useful to study since it has a myriad of potential applications in algorithm design, circuit design, image processing data compression, logic minimization, and machine learning. The report presents some background on function decomposition, genetic algorithms and results of some experiments. Although use of genetic algorithms still result in exponential growth; they provide a much lower rate of growth.			
14. SUBJECT TERMS Function Decomposition, Boolean Functions, Logic Minimization, Genetic Algorithms, Evolutionary Programming, Computational Complexity.		15. NUMBER OF PAGES 85	
		16. PRICE CODE	
17. SECURITY CLASSIFICATION OF REPORT UNCLASSIFIED	18. SECURITY CLASSIFICATION OF THIS PAGE UNCLASSIFIED	19. SECURITY CLASSIFICATION OF ABSTRACT UNCLASSIFIED	20. LIMITATION OF ABSTRACT SAR

Contents

1.0 Introduction	1
2.0 Background	3
2.1 Pattern Theory	3
2.1.1 Function Decomposition (The Simple Explanation)	4
2.1.2 Function Decomposition Example	7
2.2 The Combinatorial Problem	12
2.2.1 Partition Space	14
2.2.2 Search Techniques	16
2.2.2.1 Estimation of DFC	16
2.2.2.2 Random (Chance)	16
2.2.2.3 Decreasing Row to Column Ratio	16
2.2.2.4 Increasing Row to Column Ratio	17
2.2.2.5 Correlation	17
2.2.2.6 Genetic	17
2.3 Genetic Algorithms	17
2.3.1 Comparisons with Biological Evolution	17
2.3.2 Components of Genetic Algorithms	19
2.3.2.1 Representation of the Problem	20
2.3.2.2 Evaluating Individual Solutions ...	20
2.3.2.3 Crossover Operator	20
2.3.2.4 Mutation Operator	21
2.3.2.5 Other Parameters and Heuristics ...	21
3.0 Application of Genetic Algorithms to Partition Selection	24
3.1 Representation of the Problem	24
3.2 Evaluating Individual Solutions	26
3.3 Crossover Operator	26
3.4 Heuristics and Parameter Selection	28
3.4.1 Population Size and Setup	30
3.4.2 Crossover Rate	33
3.4.3 Mutation Rate	33
3.4.4 Generation Gap	34
3.4.5 Fitness Factor	34
3.4.6 Selection Strategy	34
3.4.7 Scaling Factor	35
4.0 Experiment Description	36
4.1 Estimation Description	36
4.2 Simulation Description	37
4.3 Summary of Experimental Results	43
4.4 Analysis of Heuristic/Parameter Variations	47
5.0 Conclusions	50
5.1 Conclusions from Experiments	50
5.2 Recommendations/Future Work	51
Bibliography	52
Appendix A. Correlation Partition Selection Algorithm Paper	53
Appendix B. Experimental Data	61

List of Figures

Figure 2.1. Diagram of a Function with N Input Variables and Table Size of $2N$.	5
Figure 2.2. Function F Decomposes to a Table Size of $2M + 2(N-M+1)$.	5
Figure 2.3. Function A Decomposes to a Table Size of $2P + 2(M-P+1)$.	6
Figure 2.4. Function B Decomposes to a Table Size of $2Q + 2(N-M-Q+2)$.	6
Figure 2.5. Both Functions A and B Decompose.	6
Figure 2.6. Decomposition Producing Two or More Functions of Table Size M.	7
Figure 2.7. Diagram of Final Decomposition of Partition in Table 2.4, Showing the inputs a, b, and d can be shared.	11
Figure 2.8. 2D Representation of Partition Space from Table 2.12.	15
Figure 2.9. 2D Representation of Partition Space from Table 2.13.	15
Figure 2.10. 2D Representation of a Typical Partition Space of an Eight Variable Function.	15
Figure 2.11. 2D Representation of the Same Eight Variable Function With Labels Rearranged to Make the Structure in Partition Space More Obvious.	15
Figure 2.12. Basic Genetic Algorithm Process.	19
Figure 3.1. Chance of Chromosome Being Selected Compared to Fitness Factor for a Population of 8.	35
Figure 4.1. Comparison of Partition Space Resulting from Full Decomposition (Left) to that Resulting from Estimation Procedure (Right).	37
Figure 4.2. Improvement of Baseline and Structured Algorithms Over Chance for Simulated Data.	40

List of Tables

Table 2.1.	A Binary Function Expressed as a String.....	4
Table 2.2.	Partition Matrix of the Function F.....	7
Table 2.3.	Partition Matrix of the Function F.....	7
Table 2.4.	Partition Matrix of the Function F.....	7
Table 2.5.	New Component Function g Created by Assigning Binary Labels to the Unique Columns in This Partition matrix.....	9
Table 2.6.	Original Function Expressed as Function of Row Variables and g.....	9
Table 2.7.	Intermediate Function h Obtained by Decomposing F Again.....	9
Table 2.8.	F Expressed as a Function of g and h.....	10
Table 2.9.	Functions g and h Created From the Partition in Table 2.4.....	10
Table 2.10.	Original Function Expressed as Function of Row Variables and New Functions.....	11
Table 2.11.	Final Decomposition of Partition in Table 2.4.....	11
Table 2.12.	Minimum Sum of Cardinality that Can Result from All Possible Initial Partition Matrices of the Original Function in the Example.....	12
Table 2.13.	Rearrangement of Labels c and d in Table 2.12.....	14
Table 2.14.	Single Crossover.....	21
Table 2.15.	Double Crossover.....	21
Table 3.1.	Two Dimensional Table Representation of the Partition Space for a Function With Six Binary Input Variables.....	25
Table 3.2.	Uniform Crossover.....	26
Table 3.3.	Structured Crossover.....	27
Table 3.4.	Six Key Differences Between the Baseline and Structured Genetic Algorithm Approaches to Partition Selection.....	29
Table 4.1.	Structured Genetic Algorithm Performance.....	42
Table 4.2.	Baseline Genetic Algorithm Performance.....	43
Table 4.3.	Structured Genetic Algorithm Performance.....	44
Table 4.4.	Baseline Genetic Algorithm Performance.....	44
Table 4.5.	Performance Comparison of Baseline and Structured Genetic Algorithm Approaches.....	44
Table 4.6.	Key Genetic Algorithm Parameters and Heuristics Analyzed in Experiments.....	48
Table 4.7.	Influence of Key Genetic Algorithm Parameters and Heuristics.....	49

Acknowledgments

There were a number of people who contributed to this report. Prof. Jim Frenzel of The University of Idaho, who provided insight to genetic algorithms and evolutionary programming techniques. Ms. Peggy Saurez provided prompt and professional secretarial support. Mr. John Jacobs, whose guidance was essential to the project. Reviews by the next level of management, Mr. Paul Johnson, Mr. Bill Moore, and Capt. Roger Williams kept the project on track. Support from the next higher level of management, Mr. Les Mcfawn was essential in allocating the resources needed for the project.

1.0 Introduction

This document reports on research into applying genetic algorithms to function decomposition in Pattern Theory. The Mission Avionics Division of the Avionics Directorate at Wright Laboratories (WL/AART) is developing an engineering design theory for algorithms, based on the structure or "pattern" of a function called Pattern Theory. The term function refers to the traditional mathematical function associating inputs with specific outputs. Pattern Theory has relevance to a number of disciplines to which these functions may relate, including Pattern Recognition, Artificial Intelligence, data compression, cryptography, and Switching Theory. Decomposed Function Cardinality (DFC) refers to the complexity of a function. The DFC is found by reducing the function to its simplest representation. Function decomposition optimizes the function with respect to the DFC using a systematic approach. The measure of DFC has been shown to correlate well with traditional measures of computational complexity such as program length, time and circuit size complexity. The ability to reduce functions to their simplest representation in the most optimal way can simplify designs, reduce computational time, and lower costs.

Function decomposition groups or partitions input variables for a function into smaller functions. Functions can be represented by two dimensional arrays, by treating the input variables as labels for the rows and columns of the arrays. These arrays are called partition matrices and several may be generated for each function by selecting different variables for the rows or columns. This selection procedure is called partition selection and can be used to reduce functions to their optimal decomposition. Among the partition selection approaches that can be used to obtain the DFC are random or chance procedures and correlation of the values of input variables and values of the function. Mission Avionics' personnel have found a more promising technique by using genetic algorithms. These algorithms use search procedures modeled after the biological process of natural selection and genetics. In the application of genetic algorithms to function decomposition, information learned from previous decompositions guides further decompositions. Genetic algorithms tend to reduce the search space and guide the search more effectively than other methods.

Section 2 of this report gives some background on Pattern Theory and its development. Also Section 2 more clearly defines Function Decomposition and DFC and gives some examples. Partition selection in the decomposition process is described, including possible techniques for optimizing the decomposition. Background is also given on

Genetic Algorithms, including the definition of heuristics and parameters used in genetic algorithms and a comparison with their biological equivalents.

The third section describes specifically the application of genetic algorithms to partition selection in the function decomposition process. This includes the representation of the problem, the techniques for performing a structured search, and the evaluation of individual solutions. A description is given of the various parameters that were varied in this program and the different heuristics which were applied. A comparison is made between the two primary genetic algorithm approaches taken to perform partition selection. One approach is referred to as the baseline genetic algorithm, following more standard genetic algorithm procedures, and the other is referred to as the structured approach. The various approaches chosen to implement these genetic algorithms to optimize function decomposition are detailed, particularly those that deviate from classic genetic algorithm procedures.

Section 4 describes experiments performed to evaluate the application of the baseline and structured search algorithms to function decomposition. Since computational time would have severely limited this program, a number of decompositions were simulated without exhaustive computer runs. The process used for this simulation is described. A comparison is made for determining the DFC between simulated decompositions and another estimation procedure.

The document concludes by offering conclusions from the experiments and then offers recommendations for the direction of future work.

2.0 Background

2.1 Pattern Theory

Pattern Theory describes an algorithm design paradigm being developed at Wright Laboratory. It attempts to help automate the algorithm design process. Pattern Theory uses a function decomposition algorithm to reduce complicated functions to their more simple components. By using function decomposition it is possible to design an algorithm that implements a function given only a partial list of inputs and outputs. Functions decompose in a number of ways. The function decomposition algorithm conducts an iterative and recursive search until the simplest decomposition is found. This search automatically designs an algorithm for reconstructing the original function using the simplest decomposition. It simultaneously provides a measure of the intrinsic complexity of the original function. This measure of complexity is called decomposed function cardinality or DFC.[1]

The idea of using the simplest representation is alternately known as the principle of parsimony or Occam's razor. The belief that the simplest explanation or representation is best has been a basic tenet of the philosophy of science. It may be the only thing that allows us to think, do math or science, or make any predictions about the world at all.

Function decomposition as used in pattern theory provides a potentially powerful technique to automate algorithm design. This has a myriad of possible applications for automated code generation, circuit design, cryptography, data compression, image processing and machine learning. This technique has been demonstrated in more than one thousand experiments with impressive results. These experiments used small binary functions of less than ten input variables.

A problem exists for functions with more input variables. The search space increases exponentially with the number of input variables. Consequently the computer time required to find the DFC becomes prohibitive. At present, there is no direct method that guarantees finding the DFC without exhaustive search. There are a number of methods that can potentially solve this search problem. Some promising techniques that limit the search space or guide the search are being investigated. Among these are genetic algorithms.

Considerable research has recently been performed in the area of genetic algorithms. They can be very useful for combinatorial hard optimization problems. This is the type

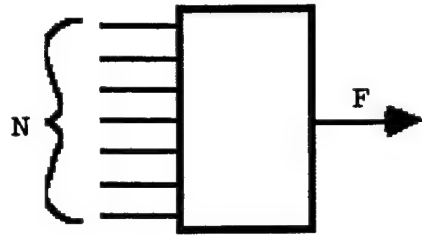


Figure 2.1. Diagram of a Function with N Input Variables and Table Size of $2N$.

Some functions can be represented as a composition of smaller functions or subfunctions. Suppose the function in Figure 2.1 can be split into two functions. M of the input variables form the input to Function A. Now the output of Function A and the remaining $N-M+1$ inputs (Function B) form the input to Function F. Two new tables are formed in the alternative representation. The sum of the cardinality of the new functions is $2^M + 2^{(N-M+1)}$. The decomposition is diagrammed in Figure 2.2.

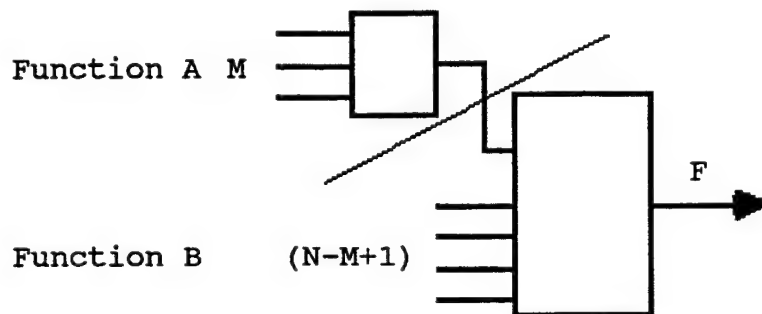


Figure 2.2. Function F Decomposes to a Table Size of $2^M + 2^{(N-M+1)}$.

A good decomposition is one that simplifies the representation and offers a cost savings, i.e., if the sum of the new cardinality is less than the original cardinality $2^M + 2^{(N-M+1)} < 2^N$. Suppose that Function A further decomposes into Functions C and D (see Figure 2.3). Figure 2.4 shows the case where Function B decomposes. If both Functions A and B decomposed the case would appear as shown in Figure 2.5. The sum of the new cardinality would be $2^P + 2^{(M-P+1)} + 2^Q + 2^{(N-M-Q+2)}$.

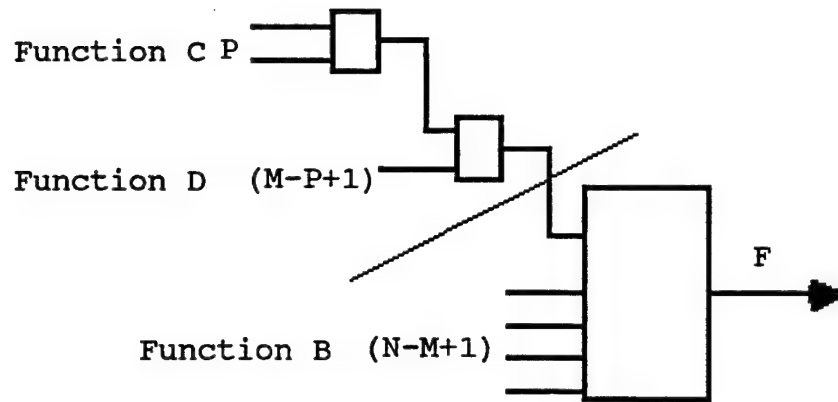


Figure 2.3. Function A Decomposes to a Table Size of $2P + 2(M-P+1)$.

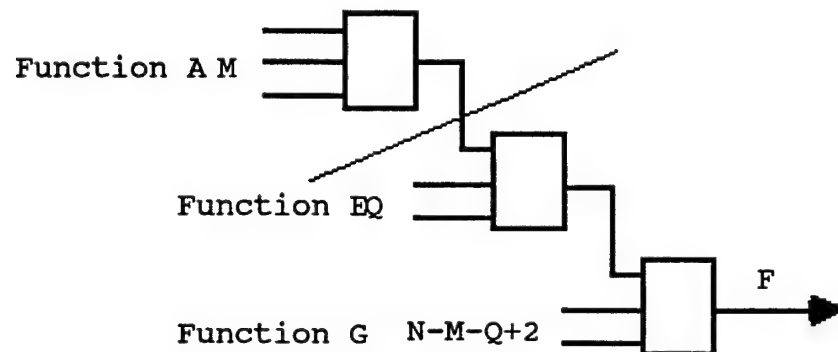


Figure 2.4. Function B Decomposes to a Table Size of $2Q + 2(N-M-Q+2)$.

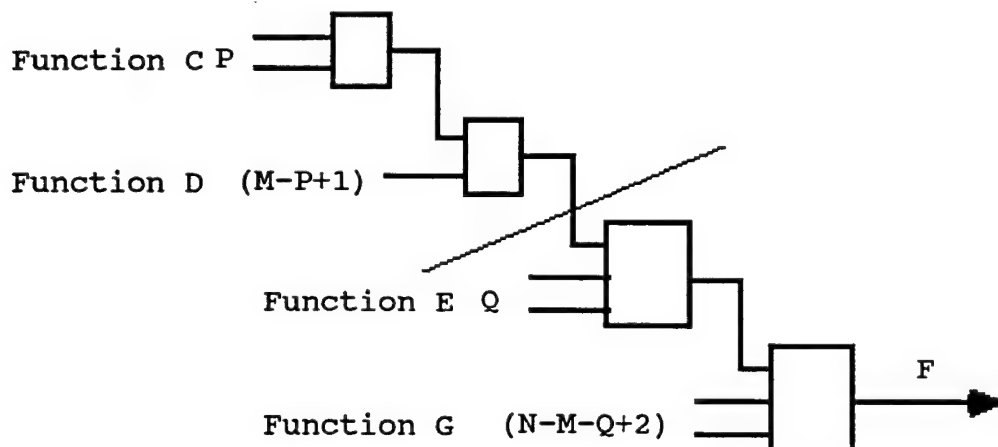


Figure 2.5. Both Functions A and B Decompose.

There is one final case to consider. Sometimes a number of variables can be removed, but more than one

function is produced. In this case two or more (m for example) tables of size M are produced (see Figure 2.6). The sum of the new cardinality is $m \cdot (2^M) + 2^{(N-M+m)}$. As long as $m \cdot (2^M) + 2^{(N-M+m)}$ is less than 2^N this is a good decomposition.

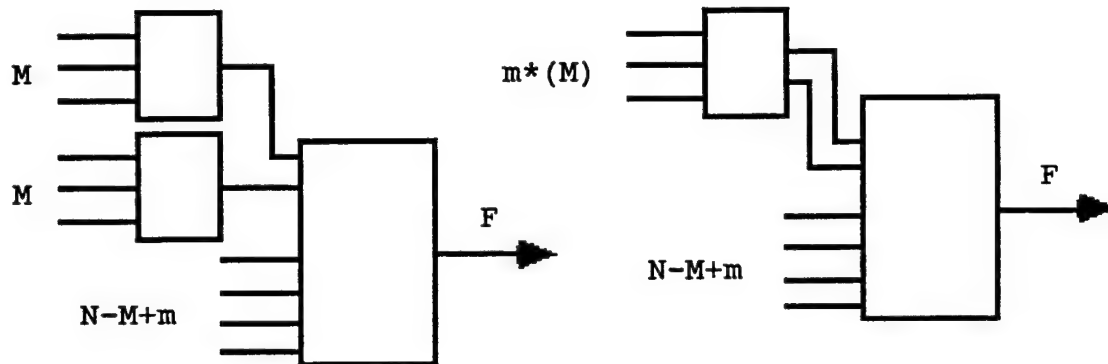


Figure 2.6. Decomposition Producing Two or More Functions of Table Size M .

2.1.2 Function Decomposition Example

The following is an illustration of the decomposition process using a binary function represented by the string or table shown previously in Table 2.1. N or the number of input variables is 6 in this case and the cardinality is 2^N or 64. The binary input variables are a, b, c, d, e and f . F is the function.

This function can also be represented by two dimensional arrays. This is accomplished by treating the input variables as labels or indices for the rows and columns of the arrays. These arrays are referred to as partition matrices. Several possible partition matrices of F are shown in Tables 2.2, 2.3 and 2.4.

Partition Matrices of the Function F .

Table 2.2.

a	00001111
b	00110011
c	01010101
def	
000	00010100
001	00101000
010	00101000
011	00000000
100	01000000
101	10000000
110	10000000
111	00000000

Table 2.3.

a	00001111
b	00110011
d	01010101
cef	
000	00000000
001	01101000
010	01101000
011	00000000
100	01101000
101	00000000
110	00000000
111	00000000

Table 2.4.

a	0000000011111111
b	0000111100001111
c	0011001100110011
d	0101010101010101
ef	
00	0001001000100000
01	0100100010000000
10	0100100010000000
11	0000000000000000

Different partition matrices are generated by selecting different input variables to label the rows and columns of the array. The selection of variables to use for row or column labels is called partition selection.

After examining the partition matrices in Tables 2.2, 2.3, and 2.4, one notices that in each partition matrix some of the columns are identical. Since some of the columns are identical, the number of unique columns in a partition matrix is less than the total number of columns. The number of unique columns in a partition matrix is called column multiplicity. Column multiplicity is important, since it can be used to determine if a function will decompose at a cost savings. If the number of unique columns can be represented by fewer binary variables than is required to represent the number of columns in the original partition matrix, then the function usually can be broken down into several smaller functions. A second condition for decomposition requires that the number of binary variables needed to represent the number of unique columns is less than the number of rows in the partition matrix.

Consider the partition matrix shown in Table 2.2. It has five unique columns. Since three binary variables are required to express decimal five, this partition matrix cannot be decomposed at a cost savings.

The partition matrix in Table 2.3 has only two unique columns. This requires one binary variable. A new component function g is created by assigning binary labels to the unique columns as shown in Table 2.5. Notice that g is a function of column variables. The original function can now be expressed as a function of the row variables and g (Table 2.6).

Table 2.5. New Component Function g Created by Assigning Binary Labels to the Unique Columns in This Partition matrix.

a	00001111
b	00110011
d	01010101
cef	
000	00000000
001	01101000
010	01101000
011	00000000
100	01101000
101	00000000
110	00000000
111	00000000
g	01101000

Table 2.6. Original Function Expressed as Function of Row Variables and g.

a	00001111	c	0000000011111111
b	00110011	e	0000111100001111
d	01010101	f	0011001100110011
g	01101000	g	0101010101010101
		F	0001010001000000

The sum of the cardinality to express g and F ($2^3+2^4=24$) is already less than the cardinality to express the original function ($2^6=64$). However, we are not done yet. We try to decompose F and g again. The function g cannot be decomposed any further, but F can be decomposed again. This time we select c, e, and f as column variables and arrive at another intermediate function h (Table 2.7).

Table 2.7. Intermediate Function h Obtained by Decomposing F Again.

c	00001111
e	00110011
f	01010101
g	
0	00000000
1	01101000
h	01101000

Finally, we express F as a function of g and h . The intermediate function g and h are functions of the input variables. Table 2.8 shows the final decomposition. Neither F , g nor h can be decomposed any further. The sum of cardinality is 20 ($2^3+2^3+2^2$) instead of 64.

Table 2.8. F Expressed as a Function of g and h .

a	00001111	c	00001111	g	0011
b	00110011	e	00110011	h	0101
d	01010101	f	01010101	F	0001
g	01101000	h	01101000		

What if there are more than two unique columns as in the partition in Table 2.4, then functions g and h would be created (see Table 2.9). Again these new functions are only a function of column variables. The functions g and h are decomposed again in Table 2.10 and the original function can now be expressed as a function of the row variables and the new functions.

Table 2.9. Functions g and h Created From the Partition in Table 2.4.

a	0000000011111111
b	0000111100001111
c	0011001100110011
d	0101010101010101
ef	
00	0001001000100000
01	0100100010000000
10	0100100010000000
11	0000000000000000
g	0100100010000000
h	0001001000100000

Table 2.10. Original Function Expressed as Function of Row Variables and New Functions.

a	0000000011111111	a	0000000011111111
b	0000111100001111	b	0000111100001111
c	0011001100110011	c	0011001100110011
d	0101010101010101	d	0101010101010101
g	0100100010000000	h	0001001000100000

Note the Xs in the final function shown in Table 2.11. These represent "don't care" conditions. There is no combination of input variables that can produce any output here, since the intermediate functions are never "one" simultaneously. The sum of cardinality for the decomposition at this level is 40 ($2^3+2^3+2^2+2^2+2^4$).

Table 2.11. Final Decomposition of Partition in Table 2.4.

a	00001111	a	00001111	c	0011	c	0011
b	00110011	b	00110011	i	0101	j	0101
d	01010101	d	01010101	g	0100	h	0001
i	01101000	j	01101000				

e	0	0	0	0	0	0	0	1	1	1	1	1	1	1
f	0	0	0	0	1	1	1	1	0	0	0	0	1	1
g	0	0	1	1	0	0	1	1	0	0	1	1	0	0
h	0	1	0	1	0	1	0	1	0	1	0	1	0	1
F	0	1	0	X	0	0	1	X	0	0	1	X	0	0

Notice that i and j are identical functions of a, b and d. They can be shared in the final decomposition as Figure 2.7 shows. The final sum of cardinality for this decomposition is, therefore, 32 ($2^3+2^2+2^2+2^4$). This point is important because it will later be used to calculate the cost of decompositions.

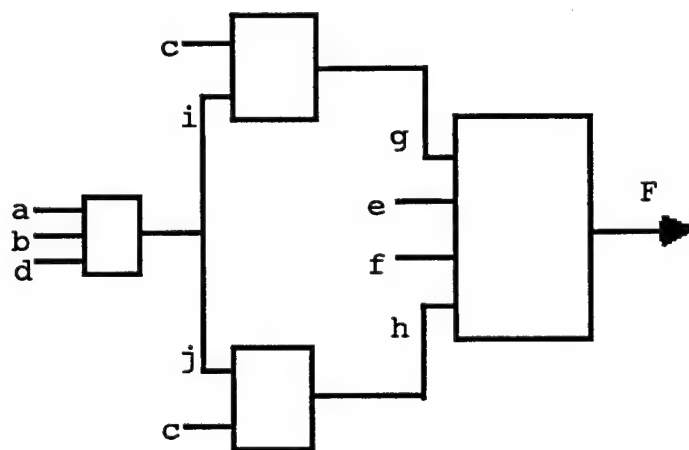


Figure 2.7. Diagram of Final Decomposition of Partition in Table 2.4, Showing the inputs a, b, and d can be shared.

Summing the cardinality of the new functions (20 in Table 2.8 and 32 in Table 2.11 with a, b and d shared) shows that both are less than the cardinality of the original

functions (64 in both Tables 2.3 and 2.4). The sum of cardinality of the new functions generated in Table 2.8 is smallest. This is an example function decomposition process. It is applied for all partition matrices and again to each of the new functions generated until the representation with the smallest sum of cardinality is found. The number of elements (cardinality) needed for this minimal representation is the measure of complexity used by pattern theory. It is called decomposed function cardinality or DFC.

2.2 The Combinatorial Problem

The number of possible non trivial partition matrices for a given binary function is the cardinality minus two. For the example, there were 62 partition matrices for the function in Section 2.1.2. Only a small number of the possible partition matrices were shown to illustrate a few points about the decomposition process. Table 2.12 shows the minimum sum of cardinality that can result from all possible initial partition matrices of the original function. Note that there are several partition selections that lead to the DFC (20) and many others that lead to some intermediate value between the original cardinality and DFC. The importance of this representation of the partition space, as it is called, will be apparent in later sections detailing the work performed in this program. The zero and one values of the binary input variables are used in Table 2.12 to designate the variables as initial row or column partition selections, respectively.

Table 2.12. Minimum Sum of Cardinality that Can Result from All Possible Initial Partition Matrices of the Original Function

a	0	0	0	0	1	1	1	1
b	0	0	1	1	0	0	1	1
c	0	1	0	1	0	1	0	1
def								
000		64	64	64	64	64	64	64
001	64	64	64	64	64	64	64	64
010	64	64	64	64	64	64	64	64
011	64	20	64	32	64	32	64	32
100	64	64	64	64	64	64	20	32
101	64	64	64	64	64	64	32	32
110	64	64	64	64	64	64	32	32
111	64	32	64	32	64	32	32	

There does not appear to be any direct method of selecting a partition that will decompose to the DFC. All the partition matrices must be examined. As mentioned, the number of possible partition matrices for a given binary function is $2^N - 2$ or the cardinality minus two. This increases exponentially as N , the number of input variables, increases. Each of these partition matrices will have a number of elements that must be compared in calculating column multiplicity. This also increases exponentially. This rapidly becomes a computationally intense problem as the number of input variables increases. The computational burden increases even more, since each new component function that is generated must also be decomposed.

The probability of finding the DFC without using any information about the function, the partition space or previously tried decompositions is no better than random draw or "chance without replacement." It's like trying to pick a black ball out of a container that has a few black balls and a large number of white ones. Initially the probability of picking a white ball is very high. As more white balls are selected the chance of getting a black ball becomes greater. The following equation illustrates this.

The probability, P , of finding the DFC on the i th try:

$$P = N_{dfc} / (N + 1 - i)$$

where:

N_{dfc} is the number of partition matrices which lead to the DFC, and

N is the total number of partition matrices.

As the search continues, the probability of finding the DFC increases. The probability, P , of finding the DFC by the i th try is:

$$P = 1 - \prod_{i=1}^{N-N_{dfc}+1} P [(N - N_{dfc} + 1 - i) / (N + 1 - i)]$$

If we had some a priori knowledge about the distribution of partition selections that led to the DFC, we could make more intelligent partition selections. If we could use information about cardinality as we made partition selections, we could improve our chances of finding a DFC more quickly. The experiments conducted in this program, which will be reported in later sections, compare some techniques for utilizing known information to make wiser partition selections. These techniques will be compared to a pure "chance without replacement" approach, which does not try to base partition selection on any previous information.

2.2.1 Partition Space

The array shown in Table 2.12 is slightly rearranged in Table 2.13 by exchanging the labels c and d. Two dimensional graphic representations of Tables 2.12 and 2.13 are shown in Figures 2.8 and 2.9 respectively. In Figures 2.8 and 2.9 a higher stack of blocks represents a better decomposition. There appears to be some form of structure in Figure 2.8. The structure becomes more obvious in Figure 2.9. The difference between Figures 2.8 and 2.9 is only a convenience of representation. There is no a priori way of determining which selection of variables for axis labels will make the structure more apparent. The structure that will be produced by any function is unknown until the decompositions are performed.

The same phenomenon is shown for a slightly more complex function in Figures 2.10 and 2.11. A typical partition space for an eight variable function is shown graphically in Figure 2.10. There appears to be some structure, but it is not obvious. Figure 2.11 shows the same function with the labels rearranged to make the structure more obvious. This representation of the function is characterized by narrow ridges and plateaus. Gradient techniques typically used for finding maxima or minima do not work well with this type of function due to the large number of discontinuities. The gradient technique is most useful when there are gradual changes leading to maxima and minima. These figures show a two dimensional representation of an N dimensional space, where N is the number of input variables.

Table 2.13. Rearrangement of Labels c and d in Table 2.12

a	0	0	0	0	1	1	1	1
b	0	0	1	1	0	0	1	1
d	0	1	0	1	0	1	0	1
cef								
000		64	64	64	64	64	64	20
001	64	64	64	64	64	64	64	32
010	64	64	64	64	64	64	64	32
011	64	64	64	64	64	64	64	32
100	64	64	64	64	64	64	64	32
101	64	64	64	64	64	64	64	32
110	64	64	64	64	64	64	64	32
111	20	32	32	32	32	32	32	

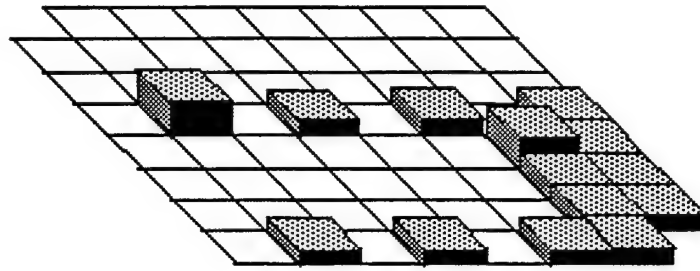


Figure 2.8. 2D Representation of Partition Space from Table 2.12.

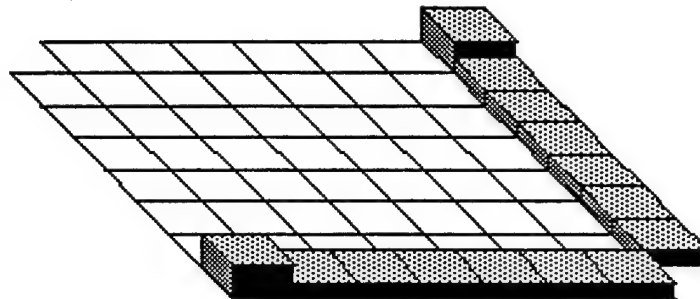


Figure 2.9. 2D Representation of Partition Space from Table 2.13.

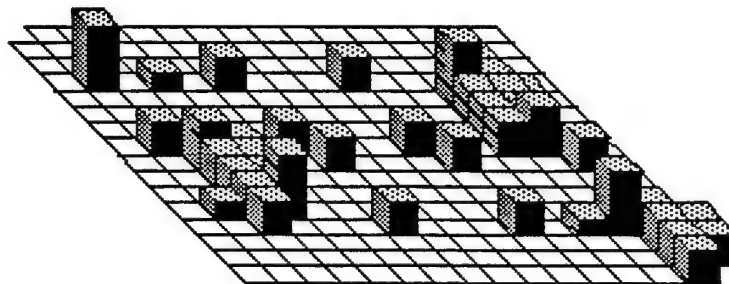


Figure 2.10. 2D Representation of a Typical Partition Space of an Eight Variable Function.

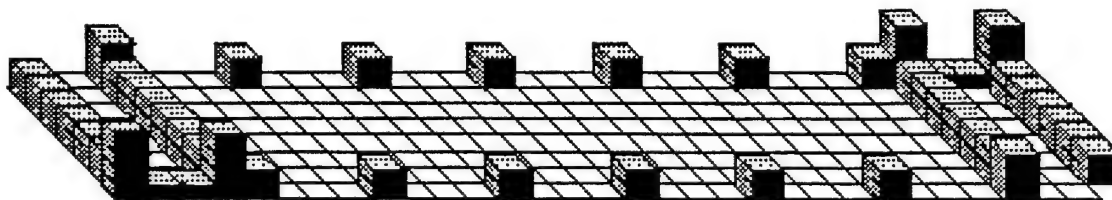


Figure 2.11. 2D Representation of the Same Eight Variable Function With Labels rearranged to Make the Structure in Partition Space More Obvious.

2.2.2 Search Techniques

A number of techniques for searching the partition space exist. They usually involve generating a partition, evaluating it, and then trying another. The problem with many of these techniques is that they use little information about the function, the decompositions that were previously found or structure of the partition space. When this information about the function is not used, it is like trying to select a good partition by pure chance. The following is a description of various techniques for searching the partition space, some which use information about the function and others that do not. Comments are also provided about their efficacy.

2.2.2.1 Estimation of DFC

Estimate the DFC based on column multiplicity or some other measure of the likelihood of a particular partition leading to a low complexity. Then only decompose partition matrices that are likely to decompose well. This relieves some of the computational burden and can be used in combination with some of the following techniques. This technique often finds the DFC or near optimal decompositions. However, it does not always find the optimal decomposition.

2.2.2.2 Random (Chance)

Randomly generate partition selections. This is obviously no better than chance. These techniques may be viable if there are a large number of partition matrices that decompose to the DFC. A variation of the chance procedure is termed "chance without replacement." "Chance without replacement" is equivalent to an exhaustive search, where nothing is done to improve the search procedure.

2.2.2.3 Decreasing Row to Column Ratio

Select partition columns first using only one variable for a column, then all combinations of variables two at a time, then three at a time and so on up to the number of inputs minus one at a time. It formed the basis for a version of the Advanced Function Decomposition algorithm used in phase one of Pattern Theory. There is no reason to believe that this is any better than chance. In fact it appears to be worse than chance. This technique will be documented in a technical report titled Pattern Theory II.

2.2.2.4 Increasing Row to Column Ratio

Select partition matrices first using only one variable for a row, then all combinations two at a time for rows, then three at a time and so on up to the number of inputs minus one at a time. There is empirical evidence that this is much better than chance. Selecting partition matrices in an increasing row to column ratio may capitalize on some underlying structure in the partition space[2].

2.2.2.5 Correlation

Perform a correlation of the values of input variables and values of the function. Then select groups of input variables with the same correlation coefficients for either rows or columns. This technique differs from those previously mentioned. It often significantly limits the search space based on information about the function. There is evidence that this technique usually leads to low complexity decompositions and is much better than chance for finding the DFC. However, there is no guarantee that any of the partition matrices selected by this method will find the optimal DFC and in some cases this method does not provide any information to guide the search. A technical paper describing this technique is provided in Appendix A.

2.2.2.6 Genetic

First, generate some decompositions by one of the previous methods. Then partially exchange some of the row or column variables in partition matrices that previously decomposed to low complexity. This should generate new partition matrices that are more likely to decompose to low complexity. This is the basic principle behind the application of genetic algorithms to partition selection. It essentially uses information about previously found decompositions. Genetic algorithms appear to be a promising technique for partition selection and were the focus of experiments conducted in this program. The next section gives some background information on genetic algorithms.

2.3 Genetic Algorithms

This section describes genetic algorithms. It gives the general flow of the algorithm. It also provides the definition, description and effects of some of the parameters and heuristics used in genetic algorithms.

2.3.1 Comparisons with Biological Evolution

Genetic algorithms are an attempt to find solutions to particularly difficult problems by using modeling mechanisms patterned after biological evolution. A collection or pool of candidate solutions to a given problem is referred to as

the population. Each member of the population, representing a particular solution to the problem, is called a chromosome. A representation of the problem space is selected that has interchangeable units or features that characterize the solution. The interchangeable units are equivalent to genes in the biological model. The specific value that a gene takes on is called an allele and often represents some characteristic of the solution.

Biologically, the chromosomes are string-like bodies containing the genes of an individual. Each gene refers to some characteristic of the individual such as eye color. The allele is the specific representation of the characteristic, such as blue or brown for the gene designating eye color. In genetic algorithms each solution or chromosome is made up of a string of bits, integers, or characters. The individual bits, integers, or characters are the genes and the specific value of each gene is the allele.

An initial population is generated by random or heuristic methods. The population is then evaluated to find the chromosomes that yield good solutions. These chromosomes are selected for survival and reproduction, usually by random draw. Chromosomes yielding less fit solutions, tend to die off. This is similar to survival of the fittest in Darwinian evolution. Alleles can be exchanged between selected chromosome pairs, as in biological crossover, or genes may be modified, as in mutation. Thus, a new population is generated. Each new population is called a generation. Each generation will contain some of the chromosomes that were good solutions from the old population and new chromosomes that should have a high probability of being better solutions to the problem. This process is repeated for each new generation until some terminating condition is reached. In each new generation the number of better solutions increases and should predominate the population. After a number of generations, the population should converge to the best solutions. A high-level flow diagram of the genetic algorithm process is provided in Figure 2.12.[2]

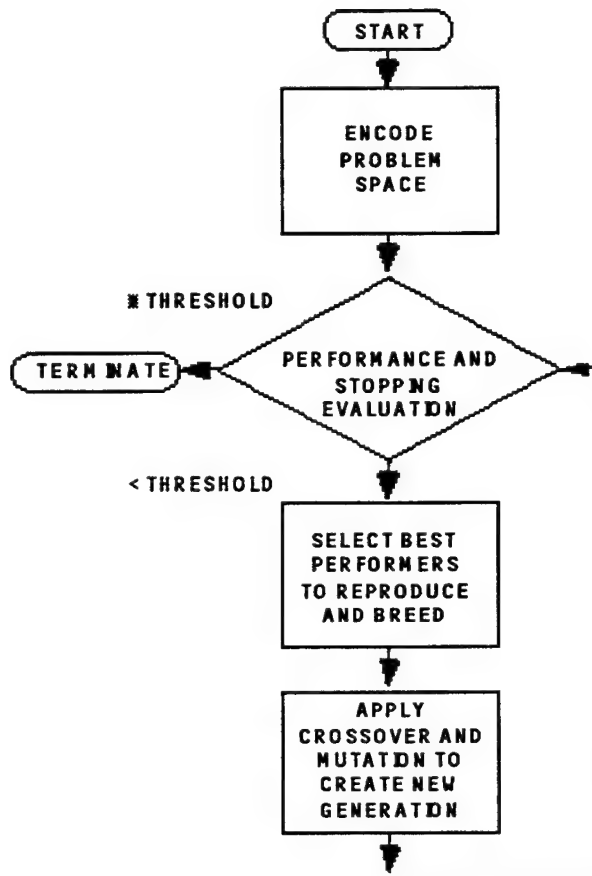


Figure 2.12. Basic Genetic Algorithm Process.

2.3.2 Components of Genetic Algorithms

There are three major components to the application of genetic algorithms: representation, evaluation, and the mechanics of reproduction. "In order to employ a genetic algorithm the analyst must supply three items: 1. The representation of the problem as a string of digits, 2. A means of evaluating individual solutions, 3. The specifics of the crossover operator." [3]

2.3.2.1 Representation of the Problem

The representation of individual chromosomes can be strings of binary digits, lists of integers, real numbers, rules or other symbols. Binary digits are considered to be the most general representation and evidence exists that the binary representation is optimal.[4,5] Representation of a problem or solution is perhaps the most formidable aspect of designing a genetic algorithm. It is one of the areas that require expertise in the problem domain and can seriously affect the performance of the algorithm. Important features of the problem must be represented in such a way that desirable solution characteristics are propagated, and undesirable characteristics suppressed.[3] The representation for chromosomes in the program discussed in this report uses binary strings with the same number of digits as the number of input variables for a function. As Section 3.0 will explain, the value of each individual allele represents an initial row or column partition selection for the corresponding binary input variable.

2.3.2.2 Evaluating Individual Solutions

Evaluation of individual solutions or determining the fitness of the chromosomes can be performed in a number of ways. If the algorithm is to perform some game or task, tournaments can be conducted with the winners or good performers selected. This is similar to competition and survival of the fittest in biologic evolution. If the algorithm is attempting to compute some function or solve an optimization problem that can be computed in some other way, then the chromosomes that are closest to the actual result are selected. This is like exploitation of an environmental niche in biologic evolution. Of course, all this implies that some independent measure of performance exists.

2.3.2.3 Crossover Operator

Before examining some details of the evaluation and selection process, such as the number of survivors or how many new members are generated, an examination of reproduction mechanisms will be performed. This examination focuses specifically on the crossover operator. Crossover is a procedure, whereby chromosome or binary string pairs exchange alleles to produce new chromosomes that may provide better solutions to the problem. Crossover techniques for a binary string are fairly standard. Single and double crossover are most commonly used in practice. These techniques are illustrated in Tables 2.14 and 2.15.

In single crossover (Table 2.14) a crossover point is randomly selected at the same position in two binary functions or chromosomes, referred to as Parents A and B. Child AB is produced by selecting the alleles in front of

the crossover point from Parent A and the alleles following the crossover point from parent B. Often a reciprocal alleleomorph Child BA is produced by selecting the alleles in front of the crossover point from Parent B and the alleles following the crossover point from Parent A.

Table 2.14. Single Crossover.

Parent A	01001010	010101001010010100	
Parent B	10010101	001010010010010100	
Child AB	01001010	001010010010010100	
Child BA	10010101	010101001010010100	

In double crossover (Table 2.15) two crossover points are randomly selected. Child ABA is produced by selecting the alleles in front of the first crossover point and in back of the last crossover point from Parent A. The alleles between the first and second crossover points are selected from Parent B. Again a reciprocal alleleomorph, Child BAB, can be produced by selecting the alleles in front of the first crossover point and in back of the last crossover point from Parent B. The alleles between the first and second crossover points are selected from Parent A.

Table 2.15. Double Crossover.

Parent A	01001	0100101	01001010010100	
Parent B	10010	1010010	10010010010100	
Child ABA	01001	1010010	01001010010100	
Child BAB	10010	0100101	10010010010100	

2.3.2.4 Mutation Operator

Typically in genetic algorithms each fundamental unit or gene has a finite probability of changing. This increases the variability of a population. In binary representations mutation usually consists of changing a bit or bits. Mutation can avoid solutions that lock on local minima and result in a search of the entire problem space.

2.3.2.5 Other Parameters and Heuristics

Once the criteria for the representation, evaluation, and crossover operator have been defined, there are still a number of parameters and heuristics that influence the algorithm's performance^[4]. These parameters often interact with each other, and the design of a genetic algorithm

becomes an iterative process. A description of these parameters is given below.

Population size: The population size chosen affects the global performance and efficiency of a genetic algorithm. A large population is desirable to insure sufficient coverage of the problem space. A small population is likely to prematurely converge on a local rather than global solution.

Crossover rate: This is defined as the percentage of the population that may be modified by the crossover operator. If the crossover rate is too high, good performers are removed faster than selection can produce improvements. If the rate is too low, the search may stagnate.

Mutation rate: The rate or probability of a mutation occurring. A low level of mutation prevents a gene from becoming locked into a particular allele and helps to insure that the entire problem space is searched. A mutation rate that is too high results in a random search.

Generation gap: The generation gap is the percentage of the population that is replaced each generation. A generation gap of one (100%) means that the entire population would be replaced. A generation gap that is too high could result in good performers being removed. The other extreme is a generation gap of zero. This would mean no replacement and is quite useless, since there would be no possibility of evolution toward a solution.

Fitness factor: This is a measure of performance of the individual chromosomes. It is often used to control both survival and reproduction. Survival and reproduction can also have separate fitness factors.

Selection strategy: The logic or heuristics that select individuals for survival and reproduction. Two common strategies are select for survival first or select for reproduction first. In select for reproduction first, all the chromosomes have a chance of reproducing. In select for survival first only those which survive reproduce.

Elitist strategy: A survival strategy which insures that the best performers survive.

Scaling factor: A scaling factor can be used to adjust the fitness factor to prevent early dominance by particular individuals, so the problem space is sufficiently explored. The scaling factor normalizes or provides a measure of relative fitness. If the scaling factor is too low the search may be near random and very slow. If this factor is too high convergence may occur at a local minimum.

Population diversity during evolution is important and a scaling factor helps maintain this diversity.

Ranking: An alternative to Fitness and scaling which ranks or sorts members of the population based on performance.

Evolutionary programming: A type of genetic algorithm which uses high mutation rates, often excludes crossover completely, a small population, elitist strategy, and a very low generation gap.

3.0 Application of Genetic Algorithms to Partition Selection

This section describes how genetic algorithms were applied to partition selection in function decomposition. Genetic algorithms seem to be a real natural for partition selection. They provide a powerful tool for finding near optimal answers in combinatorial-type problems without exhaustively searching through potentially large solution spaces. This type of optimization is precisely the problem encountered in partition selection as a search for the simplest decomposition is conducted. Specific examples why genetic algorithms provide a useful mechanism for partition selection includes the ease with which row and column selection can be represented by a binary string. Also, the cardinality in function decomposition provides a natural fitness factor that must be defined when developing a genetic algorithm. Additionally, genetic algorithms are not gradient techniques, which as mentioned earlier, do not work well with the search space in this type of problem.

In applying genetic algorithms to partition selection it was necessary, as mentioned in Section 2.3.2, to properly address the representation of the problem, the evaluation of individual solutions, and the mechanics of reproduction such as the crossover operator. The following describes the two primary approaches taken to define these characteristics as well as other genetic algorithm parameters and heuristics examined in this program. Both of the algorithms chosen used some evolutionary programming techniques. The approach referred to as the structured genetic algorithm was highly evolutionary. This algorithm varied more from methods classically used in applying genetic algorithms. The other genetic algorithm was referred to as the baseline genetic algorithm. The differences between the two approaches will be discussed with special attention given to the variations from classical methods. Several design decisions were made, because of the nature of the application, that may have limited the full evaluation of genetic algorithms in some cases. In other words, a complete parametric analysis was not performed. Section 4.0 will later provide a comparison of the results from both the baseline and structured genetic algorithm approaches.

3.1 Representation of the Problem

As previously mentioned, binary strings are favored in theory as well as in practice to represent chromosomes. In a like manner, for the partition selection problem in function decomposition, chromosomes can be represented by binary strings. Recalling the example in Section 2.2.1, Table 3.1 shows the minimum sum of cardinality that can result from all possible initial partition matrices of the original function represented. The table shown is a

representation of the partition space for a function with six binary input variables. In the genetic algorithms developed in this program chromosomes were used to specify locations in such a partition space. Each location in this space correspondingly refers to a specific initial partition selection. To specify a chromosome, each input variable (a,b,c,d,e and f in this example) is assigned a position in a binary string. The binary string has as many bits as there are input variables (six in this case). In the binary representation a "one" means the corresponding input is selected as a column variable and a "zero" means the input is selected as a row variable. This provides a very natural and compact way of defining the partition matrices to be evaluated. In the genetic algorithms when a crossover or mutation is performed the new chromosome represents a new location in the partition space. Likewise, the chromosome also represents another initial partition selection that attempts to decompose the original function to a lower cardinality.

Table 3.1. Two Dimensional Table Representation of the Partition Space for a Function With Six Binary Input Variables.

a	0	0	0	0	1	1	1	1
b	0	0	1	1	0	0	1	1
d	0	1	0	1	0	1	0	1
cef								
000		64	64	64	64	64	64	20
001	64	64	64	64	64	64	64	32
010	64	64	64	64	64	64	64	32
011	64	64	64	64	64	64	64	32
100	64	64	64	64	64	64	64	32
101	64	64	64	64	64	64	64	32
110	64	64	64	64	64	64	64	32
111	20	32	32	32	32	32	32	

In the current example, the locations in the partition space and corresponding initial partition selections that yield the DFC (lowest sum of cardinality) of 20 are represented by the following chromosomes:

Binary Variables	a	b	c	d	e	f
Chromosome 1	1	1	0	1	0	0
Chromosome 2	0	0	1	0	1	1

In one optimal initial partition selection a, b and d should be selected as column variables and c, e, and f should be row variables. In the other optimal partition selection a,

b and d should be row variables and c, e and f should be selected as column variables. A decomposition starting with these initial partition selections will lead to subfunctions having the lowest sum of cardinality possible.

3.2 Evaluating Individual Solutions

Evaluation of individual solutions, or fitness of the chromosomes, is also quite natural. It can be done by simply performing the function decomposition of the partition specified by the chromosome and obtaining the sum of cardinality. The chromosome yielding a partition with the lowest cardinality is the most fit. Column multiplicity or partially decomposing a function can also be used as evaluation factors to save the computation of doing the multi level decomposition. However, if these factors are used, the result is an estimate rather than actual cardinality. A simulated DFC and an estimate of DFC based on one level of decomposition were used as evaluation factors in this effort.

3.3 Crossover Operator

This leaves the specifics of the crossover operator. Single and double crossovers were tried in this program. Uniform crossover was also tried in the baseline genetic algorithm. The following example (Table 3.2) illustrates uniform crossover in a chromosome corresponding to a function with 26 input variables. In uniform crossover Child AB is produced by randomly selecting alleles from Parent A or B. A reciprocal allelomorph was produced by selecting alleles from the opposite parent.

Table 3.2. Uniform Crossover.

Parent A	0	1	0	0	1	0	1	0	0	1	0	1	0	1	0	0	1	0	1	0	0	1	0	1	0	0
Parent B	1	0	0	1	0	1	0	1	0	0	1	0	1	0	0	1	0	0	1	0	0	1	0	1	0	0
Child AB	1	1	0	0	0	0	1	0	0	0	0	0	1	0	0	1	0	0	1	0	0	1	0	1	0	0
alleles	b	a	b	a	a	a	a	a	b	a	a	b	b	b	b	b	a	b	a	a	b	a	b	b	b	b
Child BA	0	0	0	1	1	1	0	1	0	1	1	1	0	1	0	0	1	0	1	0	0	1	0	1	0	0
alleles	a	b	a	b	b	b	b	b	a	b	b	a	a	a	a	a	b	a	b	b	a	b	a	a	a	a

A structured crossover operator was tried (Table 3.3) in the structured genetic algorithm. This technique is named as such, because it provides a more structured search. In this form of crossover the different alleles are counted. The crossover point is not random, but each child receives approximately one half the different alleles from each parent. In each generation the number of differences is reduced which should speed up convergence of the algorithm.

Table 3.3. Structured Crossover.

Parent A	000 01000	01000101000 0010100
Parent B	000 10101	00001001001 0010100
differences	xxx x	x xx x
crossover		
Child AB	000 01000	00001001001 0010100
Child BA	000 10101	01000101000 0010100

second generation

Parent A	000 01	000 010001010000010100
Child BA	000 10	101 010001010000010100
	xx	x x
Child ABA	000 01	101 010001010000010100
Child BAA	000 10	000 010001010000010100

The genetic algorithms evaluated in this program were intended to apply to large numbers of input variables. However, to prevent population sizes from getting very large, the duplication of chromosomes was not allowed. Instead, when a mating produced a duplicate chromosome several options were exercised:

- a) Baseline Genetic Algorithm
 - o the mating was tried several more times with the same parents or
 - o a mutation was produced
- b) Structured Genetic Algorithm
 - o the duplicate chromosome was ignored and another mating was tried with different parents, otherwise the structured process would produce exactly the same children

3.4 Heuristics and Parameter Selection

Many other heuristics had to be established and parameters set to apply the genetic algorithm methodology to partition selection. Among these parameters and heuristic strategies were those defined in Section 2.3.2.4 and 2.3.2.5. The following discussion specifies how these parameters and strategies were applied to the partition selection problem. This includes the establishment of the initial chromosome population, selection strategies and rates for reproduction, and the method used to judge the fitness of the individual chromosomes. Both the baseline and structured genetic algorithm approaches are addressed. A summary of the differences between the two approaches is given in Table 3.4. A good find refers to a chromosome representing a partition that yields a new cardinality lower than the original. The following discussion will further clarify these comments.

Table 3.4. Six Key Differences Between the Baseline and Structured Genetic Algorithm Approaches to Partition Selection.

BASELINE	STRUCTURED
<p>Initial population is generated by taking first good find, compliment of 1st half and 2nd half of first good find. $Po = 3 + \#random \text{ bad finds}$</p> <p>Po: # Chromosomes in initial population.</p>	<p>Initial population is generated by taking first good find and individual compliment of each of its bits. $Po = \#bits + 1 + \#random \text{ bad finds}$</p> <p>Po: # Chromosomes in initial population.</p>
<p>Each member of population has some (tailored) probability of reproducing based on a fitness ranking.</p>	<p>Only the best reproduce in an ordered fashion.</p>
<p>Population is <u>never</u> reinitialized.</p>	<p>Population is reinitialized whenever a better chromosome is found.</p>
<p>Reproduction is via uniform crossover.</p>	<p>Reproduction is via structured single crossover.</p>
<p>Mutations occur if after N tries (usually a small number like 2 or 3) parents do not produce a new chromosome.</p>	<p>Mutation never occurs.</p>
<p>No (real) terminating condition. Search stops if solution is found or after 2^N generations</p>	<p>Terminates if all matings of "best" occur and nothing better is found. Algorithm may terminate before finding DFC.</p>

3.4.1 Population Size and Setup

To form the initial population in both the baseline and structured genetic algorithms, random chromosomes were generated until one was found, which represented a cardinality lower than the initial cardinality. All the chromosomes were stored in a population array. A population was then generated from the chromosome representing the lowest cardinality. If the array became full the "least fit" (highest cardinality) chromosomes were removed. No upper limit was placed on population size in the genetic algorithms actually evaluated. Also no tradeoff studies were performed on how population size affects convergence of the algorithm. Several methods for generating the initial population and subsequent populations were tried:

- 1) The compliment of an existing chromosome was taken to generate another member of the population.

```
00011101  Parent I
11100010  Compliment Parent II
```

- 2) In the baseline genetic algorithm the compliment of each half of the parent was taken.

```
00011101  Parent A
11101101  Child A
00010010  Child B
```

It was hoped that by doing this a few good choices would be found early. Several other variations of this method were tried. One variation included dividing the Parent chromosome into thirds and forming compliments of the various sections.

```
00 | 0111 | 01  Parent A
11 | 0111 | 10  Child C
00 | 1000 | 01  Child D
```

- 3) In the structural search algorithm the compliment of each of the input variables was taken, one at a time to generate the initial population:

```
00011101  Parent
10011101  Child A
01011101  Child B
00111101  Child C
00001101  Child D
00010101  Child E
00011001  Child F
00011111  Child G
00011100  Child H
```

The purpose of these methods was to set up an initial population, which can explore the entire space. Since the initial solution is complimented, some crossover can provide any possible combination of row/column choices. Variation 1) provides the smallest initial population, but also has the largest initial difference between chromosomes. Variation 3) has the largest initial population with the smallest difference between chromosomes.

As each new chromosome was generated, a comparison was made with existing chromosomes in the population. If it was not already in the population the new chromosome was placed at the beginning of the population array. The population was then either fully or partially sorted (ranked) by one of the methods described below.

The following description of the sort procedures uses letters to represent entire chromosomes. The numbers are fitness values that are stored with each chromosome in the genetic algorithms. The lower the fitness value, the "more fit" the chromosome.

As an example, Chromosome A may be: 010101 with a fitness value of 3

Chromosome B may be: 101101 with a fitness value of 5

1) Full Sort:

A full sort was used in which the entire population was sorted before reproduction occurred again. The following example illustrates the full sort for a population of 8 chromosomes.

Before Sort	A	B	C	D	E	F
	3	5	7	6	2	8
After Sort	E	A	B	D	C	F
	2	3	5	6	7	8

This procedure was used in the structured genetic algorithm. When a new chromosome was added, a chromosome at the end of a full population array was dropped out of the array as shown below.

	Before sort	A	B	C	D	E	F
		3	5	7	6	2	8
Chromosome	G produced:						
	Add new chromosome "4"	G	A	B	C	D	E
	Drop last chromosome "8"	4	3	5	7	6	2
	After Sort	E	A	G	B	D	C
		2	3	4	5	6	7

2) Partial Sort #1 (Compare each in sequence)

A partial sort procedure was used that begins with the first chromosome in the population. This chromosome is compared with the one directly following it in the population. If it is less fit it moves backward in the population. It is then compared with the next chromosome and the same procedure is followed. Whenever it is more fit than the chromosome directly following it, the more fit chromosome retains its position. This represents one partial sort cycle. Reproduction occurs between partial sort cycles.

Before Sort	A	B	C	D	E	F
	3	5	7	6	2	8
After Partial Sort	A	B	C	E	F	D
	3	5	6	2	7	8

This sort procedure was tried in the baseline genetic algorithm. After a number of cycles this partial sort results in a full sort. In this sort poor performers are moved right to the end of the population and do not reproduce. As in the full sort, a new chromosome at the beginning of a full population array results in the last chromosome being dropped.

	Before sort	A	B	C	D	E	F
		3	5	7	6	2	8
Chromosome	G produced:						
	Add new chromosome "4"	G	A	B	C	D	E
	Drop last chromosome "8"	4	3	5	7	6	2
	After 1st Partial Sort	A	G	B	C	D	E
		3	4	5	7	6	2
Chromosome	H produced:						
	Add new chromosome "2"	H	A	G	B	C	D
	Drop last chromosome "2"	2	3	4	5	7	6
	After 2nd Partial Sort	H	A	G	B	D	C
		2	3	4	5	6	7

3) Partial Sort #2 (Pairwise Flip)

This partial sort compares chromosomes within a pair for fitness, beginning with the first two chromosomes in the population. If the second chromosome is more fit it is flipped to the first position in the pair. After one pairwise sort through the entire population, reproduction occurs.

Before Sort	A	B	C	D	E	F
	3	5	7	6	2	8

After Partial Sort	A	B	D	C	E	F
	3	5	6	7	2	8

This sort keeps chromosomes from being moved to the end of the population too rapidly, since reproduction is based on position in population and occurs between sort operations. This procedure results in a slower sort than the others and can even get stuck if nothing were added to the population. However, new elements are placed at the beginning of the population after each partial sort. As in the other sort procedures, the last element is dropped from the population when reproduction occurs if the population array is full.

Before sort	A	B	C	D	E	F
	3	5	7	6	2	8
Chromosome G produced:						
Add new chromosome "4"	G	A	B	C	D	E
Drop last chromosome "8"	4	3	5	7	6	2
After 1st Partial Sort	A	G	B	C	E	D
	3	4	5	7	2	6
Chromosome H produced:						
Add new chromosome "2"	H	A	G	B	C	E
Drop last chromosome "6"	2	3	4	5	7	2
After 2nd Partial Sort	H	A	G	B	E	C
	2	3	4	5	2	7

3.4.2 Crossover Rate

Only two members of the population were mated each generation. Depending on the crossover scheme implemented two or four children were produced. The crossover rate was not varied from one generation to another.

3.4.3 Mutation Rate

No mutations were performed in the structured search algorithm during reproduction. However, if a more fit chromosome was generated the population was reinitialized. This procedure effectively conducts a mini search around the most fit chromosome.^[5] In the baseline algorithm mutations were performed, but not by the use of a set mutation rate. Rather by the following rule: if several attempted matings of a set of parents failed to produce a unique chromosome not in the population, a mutation occurred. This mutation consisted of changes to some of the common bits between parents.

3.4.4 Generation Gap

Since few children were produced the generation gap was very low. The generation gap was kept low to retain good solutions in the event the algorithm does not converge to optimal. Near optimal solutions are desired.

3.4.5 Fitness Factor

The fitness factor for both genetic algorithms was defined as the decomposed cardinality or DFC. Obviously, the lower the cardinality the more "fit" the chromosome or the better the decomposition.

3.4.6 Selection Strategy

The selection strategy was based on the order of the population array and always employed select for reproduction before survival. As mentioned, the population array was sorted so the most "fit" chromosomes were at the front. Conversely the last member of the array was the least "fit" after a sort. Also the last member of an array was removed each time a child was produced if the population array was full. The selection strategy depicted in Figure 3.1 was such that the probability of selecting a parent for reproduction was biased toward the start of the array. This figure shows a distribution for a population of 8 chromosomes. This selection strategy was referred to as a "tailored probability." In general the chance of being selected for reproduction improves with fitness, but not strictly. In Figure 3.1 the lower the Fitness Factor or DFC the "more fit" the chromosome. This figure represents a population with hypothetical Fitness Factors 2, 4, 3, 5, 4, 6, 6, 6.

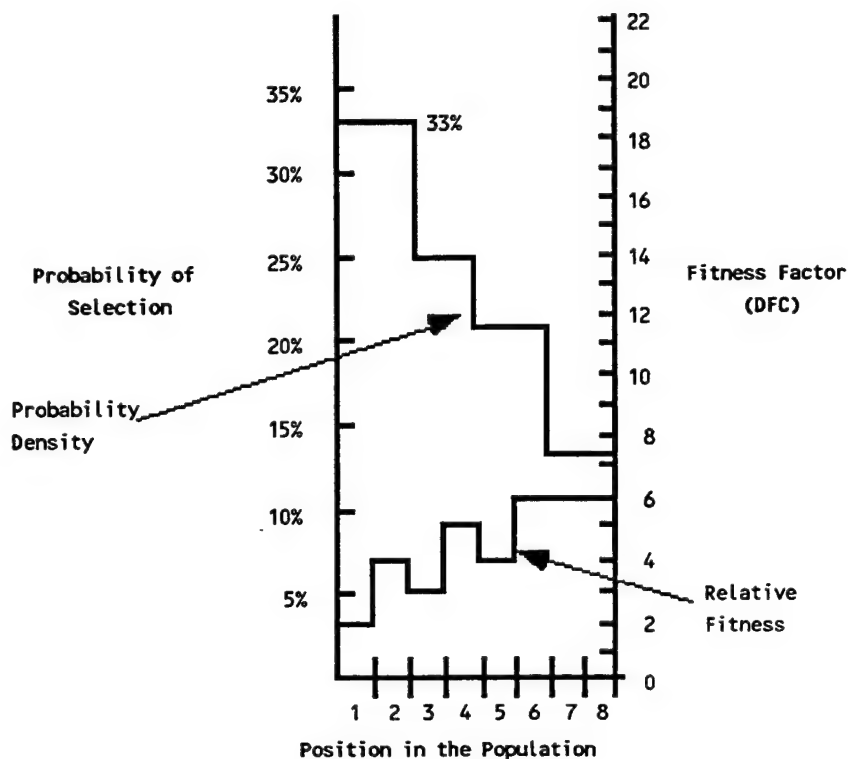


Figure 3.1. Chance of Chromosome Being Selected Compared to Fitness Factor for a Population of 8.

In the structured search each member of the population was mated in an ordered fashion. Chromosomes representing the lowest decompositions found so far were mated with other "most fit" chromosomes. Each time a "better fit" was found, the population was reinitialized. Several excursions were tried in the structured algorithm which allowed mating of slightly "less fit" members of the population. The details and effects of these excursions will be described in a later section.

In the baseline genetic algorithm several candidates were selected for reproduction and those that had the fewest different (but at least two) allele were mated. If mating had been allowed when there was only one difference, the children would be a duplicate of one of the parents.

3.4.7 Scaling Factor

The Fitness Factor was not adjusted, but as mentioned, the most fit were moved to the beginning of the population array.

4.0 Experiment Description

In a true function decomposition a function is broken all the way down to find the DFC. Since the time required to perform full function decompositions was prohibitive in this program, the DFCs were estimated or decompositions simulated. The estimation procedure attempted to predict the DFC without performing a full function decomposition. The simulation procedure attempted to simulate a complete function decomposition. It was not rigorous, but an attempt to capture some of the effects that occur while doing a decomposition. The structured genetic algorithm and the baseline genetic algorithm were evaluated in experiments by using both the simulation and estimation procedures. The following discussion describes the simulation and estimation procedures and the results of all experiments. The experimental results include comparisons of computational time to find the DFC between the two genetic algorithms and "chance without replacement." The ability to find the DFC and not some local minimum is scrutinized. Also, a comparison with chance was performed of the ability of the two genetic algorithm approaches to search increased numbers of partition matrices for increasing numbers of input variables. The influence of various genetic algorithm parameters and heuristics were also evaluated in the experiments.

4.1 Estimation Description

In the estimation procedure the algorithm would select a partition and evaluate it by performing a single level of decomposition. It would then estimate the cardinality for each partition of the function, based on column multiplicity. The cardinality was the sum of cardinality of the subfunctions carried out to only one level. In the example in Section 2.1.2 (Table 2.10) for instance, functions g and h would not have been decomposed any further. The resulting partition space would be "bumpier" (have more peaks and valleys) than if the entire decomposition had been performed. For example, recall Figure 2.9 pictured again on the left in Figure 4.1. The partition space resulting from use of the estimation procedure in comparison would appear as shown on the right in Figure 4.1.

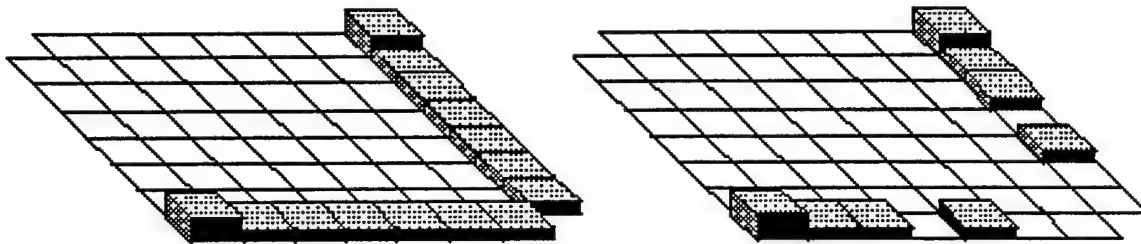


Figure 4.1. Comparison of Partition Space Resulting from Full Decomposition (Left) to that Resulting from Estimation Procedure

The estimated data could result in what is referred to as a positive decomposition. This results if the estimated cardinality is higher than the original cardinality. In the algorithm this positive decomposition could be allowed or disregarded for experimental analyses. If the algorithm was set to suppress the positive decomposition then the original cardinality was used as the estimate. The estimation procedure was tried for 26 functions in the experimental analysis. Also, for the estimation procedure, only eight variable functions were used.

4.2 Simulation Description

In the simulation procedure the user decides which sets of input variables must be grouped in columns, in rows, or can be mixed. The algorithm then attempts to find the row/column assignments. It makes trial row and column selections by breaking out certain variables as subfunctions and calculates a value that corresponds to the cardinality for each row/column selection. This calculation is based upon the discussion in Section 2.1.1. The new cardinality is compared to the original cardinality. Three particular cases apply in the simulation procedure when making row/column partition selections.

CASE 1:

If a set of input variables is specified to appear entirely in rows and the algorithm selects them as such: divide cardinality by $2^{(M-1)}$ and add 2^M to the cardinality for each input of M variables where this occurs.

- Otherwise the cardinality remains the same as the original cardinality.

CASE 2:

If a set of input variables is specified to appear entirely in columns and the algorithm selects them as such: divide cardinality by $2^{(M-1)}$ and add 2^M to the cardinality for each input of M variables where this occurs.

- Otherwise the cardinality remains the same.

CASE 3:

If a function can appear partially in rows or columns and the algorithm selects them as such: divide cardinality by $2^{(M-2)}$ and add $2^{(M-2)}$ to the cardinality for each input of M variables where this occurs. This is analogous to the $m \cdot (2^M) + 2^{(N-M+m)}$ example in Figure 2.6, where $m=2$.

- If the function appears entirely in rows or columns the cardinality is calculated by the rules in Case 2 or 3.

There are other conditions that must be satisfied or the original cardinality is returned:

- 1) At least one variable must be a row variable.
- 2) At least one set of input variables must be entirely in columns.
- 3) The final cardinality must be less than the original cardinality.

In the baseline genetic algorithm positive decompositions could be allowed if they occur or such decompositions could be suppressed. Positive decompositions were always suppressed when the estimated data was used. Again, a positive decomposition refers to a decomposition where the new cardinality is larger than the original cardinality. With positive decomposition suppressed no cardinality larger than the original cardinality is used as an estimate. No excursions were tried with the crossover operator in the baseline genetic algorithm.

In the structured genetic algorithm positive decompositions were also allowed for the simulated data. Excursions were also tried with the crossover operator later in the program to try to improve the search and find more DFCs. In some cases the compliment of the initial population or $P[0]$ compliment was taken of the best performers after the initial search terminated. Observation of other trials showed that sometimes the DFC occurred at the compliment of the best cardinality chromosomes found. These trials did result in finding more DFCs. In some cases all of the partitions leading to the DFC were found. Another crossover operator excursion was called the "odd child fix." This excursion was tried in the structured search algorithm, producing 4 children if the number of different alleles was odd between two mating chromosomes. This excursion was used because observation of a number of trials seemed to indicate the search will take longer if all possibilities are not examined initially. In this crossover scheme two crossover points were selected as shown in the example below. The "odd child fix" improved speed significantly and performance (ability to find the actual DFC) marginally on the simulated data. In cases where there

were an even number of differences between the parents then structured crossover was used.

Parent A xxxxx
Parent B yyyyy

Child C xxyyy
Child D yyxxx The old method stopped here
after two children.

Child E xxxyy
Child F yyyxx The "odd child fix" added two
children.

4.3 Summary of Experimental Results

Experimental results were analyzed to see if improvements over "chance without replacement" were realized by using the structured or baseline algorithms. An improvement was judged to be a reduction in the number of partition matrices searched before the DFC was discovered. Appendix B is a comprehensive table listing all the experimental results. Figure 4.2 summarizes experimental data from the performance comparison of the baseline and structured genetic algorithm approaches. These results are taken from trials using simulated data to identify the DFC. The curves shown are averages from typical experiments using a number of functions. The data show the number of initial row/column selections made before the DFC was found as a function of the number of input variables. As Figure 4.2 shows, both the structured and baseline algorithms reduce the search space compared to the random or chance approach. As the number of input variables are increased, the effectiveness of the genetic algorithm approaches are magnified. The structured approach in particular dramatically reduces the number of partition matrices that are examined before the DFC is found. The experimental results showed that the structured search algorithm can find an answer up to 10 times faster than exhaustive search for 8 bit functions with simulated data. However, the cardinality found is sometimes not the minimum cardinality (DFC).

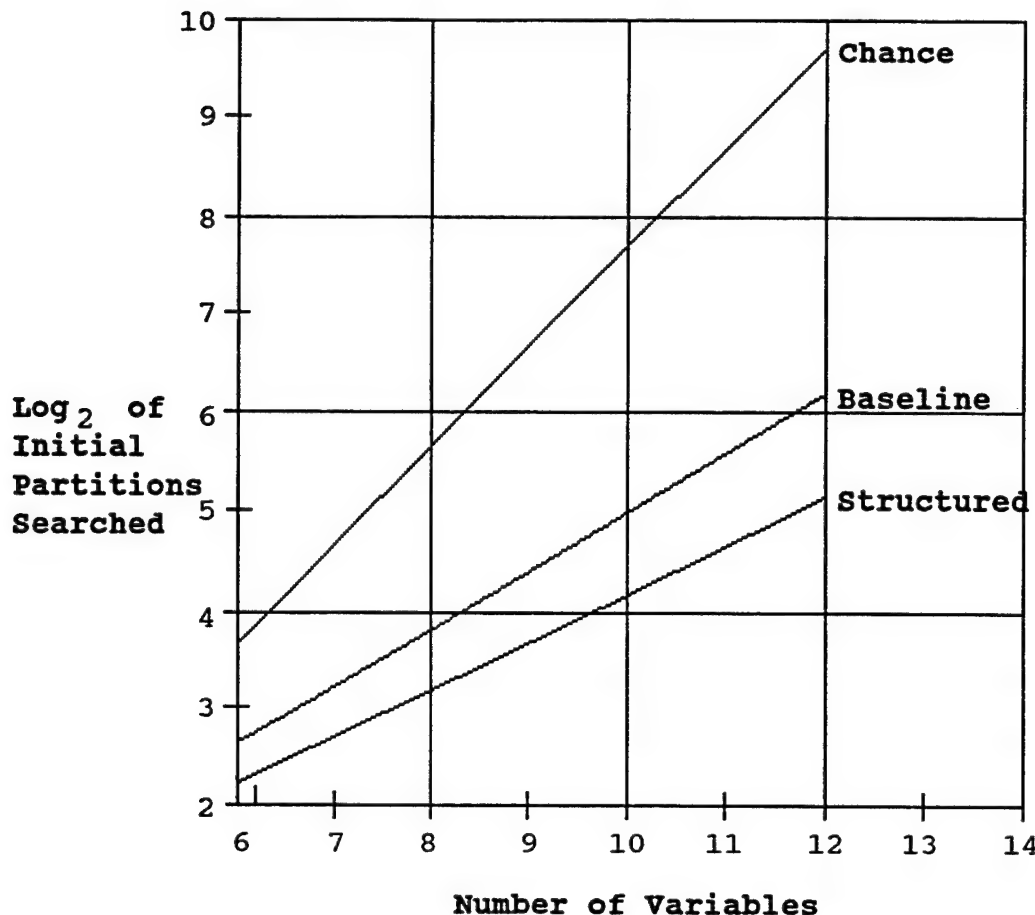


Figure 4.2. Improvement of Baseline and Structured Algorithms Over Chance for Simulated Data.

Tables 4.1 to 4.4 provide a detailed representation of the experimental results for estimated data. The performance of the baseline and structured algorithms are compared to "chance without replacement." A comparison can also be made between the performance of the two algorithms. The influence of allowing or suppressing positive decompositions on finding the DFC is examined. Also shown is the effect of taking the compliment of the initial population and using the "odd child fix" technique in the structured algorithm. Tables 4.1 and 4.2 provide results for all the functions evaluated by each algorithm. The last two tables present the results for functions which only have one partition leading to the DFC. This second comparison was made to see if the experimental results would look any different for the most difficult searches.

Tables 4.1 to 4.4 are actually two-dimensional arrays. The columns show the percentage of the time the algorithms performed better or worse than "chance without replacement" (see Section 2.2.2.2). Better than chance means the answer

was found faster than by using "chance without replacement" procedures. The computational speed refers to the reduction in the partition space that had to be searched by the new algorithms compared to "chance without replacement." For example, 4x the speed means that only 1/4 of the partition matrices had to be searched as compared to an exhaustive search ("chance without replacement"). Obviously, this reduction in search space will also reduce computational time significantly. The rows refer to the ability to find the actual DFC. The row marked "DFC Found" refers to the percentages when the actual DFC was found. "DFC Not Found" refers to the percentage of time when something other than the DFC was found. Recall that the structured search algorithm has a termination condition that can result in the termination of a search before the DFC is found (see Table 3.4).

Table 4.1. Structured Genetic Algorithm Performance All Functions.

With Positive Decomposition

	% Time Better Than Chance	% Time Worse Than Chance	Computational Speed Improvement Over Chance
DFC Found	41.42	6.82	7.14x
DFC Not Found	51.43	0.34	

With Positive Decomposition Suppressed

	% Time Better Than Chance	% Time Worse Than Chance	Computational Speed Improvement Over Chance
DFC Found	55.23	12.34	4.47x
DFC Not Found	32.2	0.23	

With Positive Decomposition and P[0] Compliment

	% Time Better than Chance	% Time Worse than Chance	Computational Speed Improvement Over Chance
DFC Found	63.80	3.62	4.4x
DFC Not Found	32.25	0.34	

With Positive Decomposition and "Odd Child Fix"

	% Time Better than Chance	% Time Worse than Chance	Computational Speed Improvement Over Chance
DFC Found	64.15	7.15	4.29x
DFC Not Found	28.34	0.35	

Table 4.2. Baseline Genetic Algorithm Performance All Functions.

With Positive Decomposition

	% Time Better than Chance	% Time Worse than Chance	Computational Speed Improvement Over Chance
DFC Found	69.69	30.31	1.5x
DFC Not Found	0.0	0.0	

With Positive Decomposition Suppressed

	% Time Better than Chance	% Time Worse than Chance	Computational Speed Improvement Over Chance
DFC Found	71.85	27.23	1.56x
DFC Not Found	0.0	0.92	

Table 4.3. Structured Genetic Algorithm Performance
Functions With One Decomposition or Partition Leading to the
DFC.

With Positive Decomposition

	% Time Better than Chance	% Time Worse than Chance	Computational Speed Improvement Over Chance
DFC Found	50.84	4.51	11.49x
DFC Not Found	44.51	0.15	

With Positive Decomposition Suppressed

	% Time Better than Chance	% Time Worse than Chance	Computational Speed Improvement Over Chance
DFC Found	58.15	0.0	6.85x
DFC Not Found	41.75	0.11	

With Positive Decomposition and P[0] Compliment

	% Time Better than Chance	% Time Worse than Chance	Computational Speed Improvement Over Chance
DFC Found	69.13	0.04	7.35x
DFC Not Found	30.58	0.25	

With Positive Decomposition and "Odd Child Fix"

	% Time Better than Chance	% Time Worse than Chance	Computational Speed Improvement Over Chance
DFC Found	69.13	0.04	7.15x
DFC Not Found	30.65	0.18	

Table 4.4. Baseline Genetic Algorithm Performance Functions
With One Decomposition or Partition Leading to the DFC.

With Positive Decomposition

	% Time Better than Chance	% Time Worse than Chance	Computational Speed Improvement Over Chance
DFC Found	87.45	12.55	2.19x
DFC Not Found	0.0	0.0	

With Positive Decomposition Suppressed

	% Time Better than Chance	% Time Worse than Chance	Computational Speed Improvement Over Chance
DFC Found	82.73	16.55	2.25x
DFC Not Found	0.0	0.73	

These tables show several significant findings from the genetic algorithm performance comparison. In the structured algorithm allowing positive decompositions led to increased computational speed over suppression of positive decompositions, but the algorithm did not locate the DFC as well. Adding the P[0] compliment or "odd child fix" routines slowed down the computational speed, but improved the chance of finding the DFC. In the baseline search positive decompositions did not make a drastic difference. Computational speed improvement over chance was still significant for the baseline algorithm, but not as much as the structured search. However, the baseline algorithm found the DFC virtually every time, while the structured search located local minima a fair amount of the time. For functions with only one partition leading to the DFC, the structured and baseline genetic algorithms performed even better, compared to chance, than for functions with multiple DFCs. In other words, the more difficult the search the more benefit the genetic algorithms could provide. The structured approach showed less dependence on the type of function.

Recall that the DFCs were identified in this experimental program by either a simulation procedure or estimation procedure, they were not located by performing full function decompositions. Since there were differences between these two procedures, the performance comparison of the algorithms showed dependence on the particular procedure used to identify the DFC. Table 4.5 summarizes some of the findings from the algorithm performance comparison in relation to the estimation or simulation procedures. Both

algorithms were an improvement over chance in the time required to locate the DFC, whether the estimation or simulation procedure was used. However, as the table states, both algorithms sometimes identified a local minimum rather than the DFC for the function. This occurred much more often using the estimation procedure or estimated data. Using the simulated data, both algorithms were much more apt to locate the actual DFC. The table also shows that the baseline algorithm was better able to locate the actual DFC when compared to the structured algorithm. This was true whether the estimated data or simulated data was used.

Table 4.5. Performance Comparison of Baseline and Structured Genetic Algorithm Approaches.

BASILINE	STRUCTURED
Speed improvement over chance.	Speed improvement over chance. Less function dependent variance.
Rarely gets stuck on a local minimum using simulated data.	Rarely gets stuck on a local minimum using simulated data, but gets stuck more often than baseline.
Often gets stuck on a local minimum using estimated data.	Gets stuck more often than baseline on a local minimum using estimated data.

4.4 Analysis of Heuristic/Parameter Variations

Although an exhaustive parametric study was not conducted, variations in some of the genetic algorithm parameters and heuristics did allow some conclusions to be assessed from the experimental results. Many of the parameters and heuristics were defined differently between the baseline and structured genetic algorithm approaches, making their influence often apparent. Table 4.6 describes the different parameters and heuristics evaluated for each of the two algorithms. It also shows which parameters were examined in detail and which were tried, but for one reason or another were not used in the experiments. The initial population was either large, using many chromosomes, or limited, starting out with fewer chromosomes. The crossover operators used were described previously. Mutation was only used for the baseline algorithm in some cases. Reproduction rules were used as described earlier. Tailored probability refers to the selection strategy shown in Figure 3.1. "Best" refers to cases where only the best performers at the front of the sorted population array were reproduced. "N-best" refers to cases where some of the best lower performing chromosomes, but not the worst performers, were allowed to reproduce. Real uniform refers to cases where chromosomes were selected uniformly throughout the population for reproduction. Table 4.7 illustrates the influences of these different parameter and heuristic settings on the performance of each algorithm.

Table 4.6. Key Genetic Algorithm Parameters and Heuristics Analyzed in Experiments.

Initial Population	
Baseline	Structured
Few	Few
Many	Many

Crossover	
Baseline	Structured
Single	Structured Single
Double	Double
Higher Order	Higher Order
Uniform	

Mutation	
Baseline	Structured
None	None
Rate	
Rule	

Mating	
Baseline	Structured
Uniform	Ordered
Tailored Probabilit y	Best
N-best	N-best
Real Uniform	

Legend

No Shading

Not Tried

--

Tried, But Not
Used

--

Used For Data In
Experiment

Table 4.7. Influence of Key Genetic Algorithm Parameters

Crossover	<ul style="list-style-type: none"> o Uniform crossover tends to work best & fastest. It worked best in baseline search. o Single or double crossover takes longer. Single crossover used with structured search tends to increase time in baseline search. o Using the structured crossover operator with the baseline search to generate an initial population results in the search getting stuck in a local minimum.
Mutation	<ul style="list-style-type: none"> o If mutation is not used searches will get stuck in a local minimum as literature predicts. Trying some mutation in structured search might help.
Selection Strategy for Reproduction	<ul style="list-style-type: none"> o Tailored probability seemed to work best in baseline search. o Limiting mating to N-best in baseline search caused some local minimum problems. o Using something other than tailored probability for choosing parents in the baseline search resulted in a longer search. o In the structured search mating using N-best ($N > 1$) relieved some of the local minimum problems, but caused it to take much longer.

5.0 Conclusions

5.1 Conclusions from Experiments

The experiments conducted in this program on the baseline and structured genetic search algorithms showed very promising results. Both algorithms performed faster than chance overall. This is significant since it points out that elements of both algorithms can be used to reduce computational time for function decomposition. Speed is important, but the ability to find the DFC is vital. Both algorithms showed the ability to find the DFC and yet be faster than chance. Experiments showed that the structured search was faster than the baseline, but did not always find the DFC using estimated data, because of a restrictive reproduction and termination condition. The structured search, however, always found the DFC using the simulated data. A terminating condition that needs to be included in the baseline algorithm will likely reduce some of its ability to find the DFC. The accurate performance of these algorithms for various combinations of heuristics and parameter settings, along with an increase in speed over chance, warrants a further investigation of their applicability.

The structured search algorithm can be tailored to include additional heuristics that could improve its performance. Several excursions were tried in this program, as mentioned previously, such as allowing mating of slightly "less fit" decompositions. These excursions, however, were found to only increase the time required for convergence in the simulated data. They were not tried on the estimated data.

Both algorithms showed improvement over chance in the number of partition matrices searched, as the number of variables increased. The rate of growth of the search space is still exponential, but has a slower rate of increase. This is very promising, since as the number of variables increases, the time required to evaluate a partition also increases exponentially.

Regrettably, time did not allow a full parametric analysis of the genetic algorithm approaches. However, the work performed up to this point looks promising. Both algorithms can provide a significant speed increase in finding the DFC. Additional improvements can probably be attained by using a better method of selecting the initial population, i.e., correlation techniques and/or increasing the number of rows in the initial population (see Section 2.2.2). Both algorithms would need to be tailored depending on the application.

5.2 Recommendations/Future Work

For limitations that were found, several possible fixes could be studied to potentially solve the problems. Other genetic algorithm variations could be investigated, including both standard and nonstandard practices. The particular problems that were encountered could be studied more to identify particular heuristics or theories that may prove useful. Some were tried in this program already. Additional work could identify better ways to generate the initial population. The P[0] compliment, for example, could be applied to more of best performers than just the top ones.

Some of the open issues that may require additional investigation include the presence of a large number of discontinuities in the estimated data. The estimated data may not have provided a very accurate estimate of the DFC. Obviously, it would be beneficial to try both algorithms on actual data, or in other words allow a function to be completely decomposed to the DFC. A comparison could then be made between the actual data and the estimated and simulated data. A program called FLASH (Function Learning and Synthesis Hotbench) resides at the Avionics Directorate that carries out decompositions more thoroughly than those algorithms used in this study. This program could be applied to functions examined in this study to verify the effectiveness of the structured and baseline algorithms.

Bibliography

- [1] Timothy D. Ross, Michael J. Noviskey, Timothy N. Taylor, David A. Gadd, Pattern Theory: An Engineering Paradigm for Algorithm Design, Final Technical Report WL-TR-91-1060, Wright Laboratories, USAF, WL/AART, WPAFB, OH 45433-6543, August 1991.
- [2] Scott Austin, An Introduction to Genetic Algorithms, AI Expert, March 1990.
- [3] Kendall E. Nygard, Rhonda K. Ficek, Ramesh Shadra, Tools of the Trade: Genetic Algorithms, OR/MS Today, August, 1992.
- [4] John H. Holland, Genetic Algorithms, Scientific American, July 1992.
- [5] David M. Levine, A Genetic Algorithm for the Set Partitioning Problem, Proceedings of the Fifth International Conference on Genetic Algorithms, 17-21 July 1993.

Appendix A.

Correlation Partition Selection Algorithm Paper

Michael J. Noviskey

30 Aug 92

1.0 Introduction

This memo documents work done on a partition selection algorithm for function decomposition as described in Ref 1. This memo includes a brief description of the need for partition selection algorithms; a candidate algorithm for generation of coefficients corresponding to the input variables of a function; how to use the coefficients in selecting rows and columns of a partition matrix; and several examples and observations about this algorithm.

2.0 The need for partition selection

In order to decompose a function the Ada Function Decomposition (AFD) algorithms essentially perform an exhaustive search, with a few limiting heuristics. There are about two raised to the number of input variables partition matrices to be searched for the original function. If the function decomposes, more functions of lower complexity are produced for further decomposition. This process is continued until functions of the lowest complexity are found. No attempt was made to select partitions that would decompose to some low cost initially. If this were done it could limit the search space or prune the search tree later on. No attempt was made to develop heuristics which would predict good partitions for decomposition. These choices were made in order to meet some of the objectives of Pattern Theory I. But in order to develop a practical algorithm, some type of partition selection algorithm which can guide the search needs to be developed.

2.1 Algorithm description

The algorithm description is divided into two main parts. The first is a description of generation of coefficients which correspond to input variables. The second is utilization of the coefficients to select row and column variables for the partition matrix.

2.1.1 Coefficient Generation

For each defined value of the function a coefficient for each input variable is incremented if the value of the input variable matches the value of the function. The coefficient is decremented if the input variable does not match the value of the function. The following BASIC subroutines provide a detailed implementation. These routines assume that true vacuous variables have been removed. If vacuous variables are not

explicitly removed as a preliminary step, they will increase the run time by increasing the size and number of partition matrices which must be considered.

Two routines not documented here are ZeroVector and GetBinary. ZeroVector sets the elements of a one dimensional array to zero. And GetBinary returns a binary vector representation of a base ten argument. In this case the vector returned from GetBinary represents the input to the function.

```

SUB GenCoefs (Function$, A%(), X%())
'
' The purpose of this routine is to generate partition
selection coefficients.
'
' Function$  =>  A string representation of a Function.
'
' X$         =>  The Nth value of the function
'              {0,1,x}
'
' X%()       =>  Binary vector representation of input
for the      Nth value of the function.
'              Vector of {0,1}
'
' A%()       =>  Partition selection coefficients.
'              Vector of integers
'
      ZeroVector A%():
selection
' First set partition
' coefficients to zero.
      FOR N% = 1 TO LEN(Function$):
the function:
' For each element of
      X$ = MID$(Function$, N%, 1):
' Get it's value.
' If this element of
the function
' is defined {0,1}.
      IF X$ = "0" OR X$ = "1" THEN
input vector.
      GetBinary (N% - 1), X%():
' Get it's binary
      GenCoef VAL(X$), A%(), X%():
' Modify the
Coefficients.
      ELSE
' If this element of
the function
' is not defined then
do nothing.
      END IF
      NEXT N%
END SUB

```

```

SUB GenCoef (Value%, A%(), X%())
'
'   The purpose of this routine is to increment or
decrement
'   partition selection coefficients depending on whether the
'   associated input matches the Nth Value of the function.
'
'   Value%   =>   The Nth value of the function
'               {0,1}
'
'   X%()     =>   Binary vector representation of input for
the
'               Nth value of the function.
'               Vector of {0,1}
'
'   A%()     =>   Partition selection coefficients.
'               Vector of Integers
'
'   For each input:
'   FOR I% = LBOUND(X%) TO UBOUND(X%)
value
'               If it matches the
'               of the function
'
'               IF X%(I%) = Value% THEN
'                   A%(I%) = A%(I%) + 1:
coefficient;
'                   increment it's
'               ELSE :
'                   otherwise
'                   A%(I%) = A%(I%) - 1:
coefficient.
'                   decrement it's
'               END IF
'               NEXT I%
END SUB

```

2.1.2 Coefficient Utilization

The above procedure will produce coefficients associated with the input variables. Selecting groups of coefficients of the same magnitudes as column or row variables can result in a good choice for a partition matrix. Since coefficients of the same absolute value may or may not result in a decomposition, combinations within a group must also be tried. However, combinations across groups do not appear to need to be tried. At least no case has been found, where combinations across groups resulted in a decomposition.

To illustrate this procedure, suppose we have three sets of coefficients a_1, a_2, a_3 ; b_1, b_2 ; and c_1 . Where a_1, a_2 , and a_3 are of the same absolute value; b_1 and b_2 are of the same absolute value. The following combinations must be tried as column variables, again assuming vacuous variables have been removed. Combinations which are tried as column choices are shown between the [] brackets.

```

[(a1, a2, a3), (b1, b2)], [(a1, a2, a3), (b1)],
[(a1, a2, a3), (b2)],
[(a1, a2, a3), (c1)], [(a1, a2, a3)],
[(a1, a2), (b1, b2), (c1)],
[(a1, a2, a3), (b1), (c1)], [(a1, a2, a3), (b2), (c1)],
[(a1, a3), (b1, b2), (c1)], [(a2, a3), (b1, b2), (c1)],
[(a1, a2), (b1, b2)], [(a1, a3), (b1, b2)],
[(a2, a3), (b1, b2)],
[(a1, a2), (c1)], [(a1, a3), (c1)], [(a2, a3), (c1)],
[(a1), (b1, b2)], [(a2), (b1, b2)], [(a3), (b1, b2)],
[(a1), (c1)],
[(a2), (c1)], [(a3), (c1)], [(b1, b2), (c1)], [(b1), (c1)],
[(b2), (c1)], and [(b1, b2)].

```

Notice that combinations like $((a_1, a_2), (b_1))$ were not tried, since no case has been found where choosing some elements from one group and some from another results in a good decomposition. This results in twenty six rather than fifty six partitions (not testing for vacuous variables) which would have been required to perform an exhaustive search of a six variable function. The percent reduction in search space is more dramatic as the number of variables increases, but it depends on the number of groupings and number of elements within a group. The worst case occurs either when all the coefficients are the same or all are different. Then the entire space must be searched again. The best case occurs when the number of groups is about equal to the number of elements in the groups. The treatment of the actual number of partition matrices which must be tried as a function of number of groupings and grouping size is a little more complex and still needs to be addressed.

2.2 Examples

2.2.1 A very simple example

This example illustrates the column selection procedure on a very simple function of three variables. Initially A_1, A_2 and A_3 are incremented, since X_1, X_2 and X_3 all match the first value of the function. A_1 and A_2 are incremented again since X_1 and X_2 match the second value of the function, but A_3 is decremented since X_3 does not match the function value. This process is repeated for the entire function until the final values 2, 6 and 2 are found for A_1, A_2 and A_3 . (see figure below). Since the coefficients for A_1 and A_3 have the same magnitudes, this suggests that X_1 and

X3 should be grouped. Below are three of the partition matrices which can be formed from the function. Note that when X1 and X3 are selected as column variables, column multiplicity, Nu, is minimized. This function

X1	X2	X3	F	A1	A2	A3
0	0	0	0	1	1	1
0	0	1	0	2	2	0
0	1	0	0	3	1	1
0	1	1	1	2	2	2
1	0	0	0	1	3	3
1	0	1	0	0	4	2
1	1	0	1	1	5	1
1	1	1	1	2	6	2

could be broken down into F1 (F2 (X1, X3), X2), although no cost savings would occur for a three variable function.

X2 & X3				X1 & X3				X1 & X2						
X1	0	0	0	1	X2	0	0	0	0	X3	0	0	0	1
	0	0	1	1		0	1	1	1		0	1	0	1
Nu = 3				Nu = 2				Nu = 3						

2.2.2 A Non Decomposable Function

The partition matrices shown for the four variable function shows that it will not decompose. Yet the partition selection algorithm will generate different coefficients.

X1 & X2					X1 & X3					X1 & X4							
X3 &	1	0	1	1	X2 &	1	0	1	0	X2 &	1	0	0	0			
	1	1	0	1		1	1	0	1		1	1	0	0			
X4	0	0	0	0	X4	0	0	1	1	X3	1	1	1	1			
	1	1	1	0		0	0	1	0		0	1	1	0			
X1 & X2 & X3										X1 & X2 & X4							
X4	1	0	1	1	1	1	0	1	X3	1	0	1	1	0	0	0	0
	0	0	0	0	1	1	1	0		1	1	1	1	0			
X1 & X3 & X4										X2 & X3 & X4							
X2	1	0	1	1	0	0	1	1	X1	1	1	1	0	0	0	1	1
	1	1	0	1	0	0	1	0		0	1	0	0	1	0		

The coefficients generated by the partition selection algorithm were -2, -2, 6, -6 respectively. This suggests that the first matrix or one of the last four would give a good decomposition. Yet no decomposition works.

2.2.3 Absolute vs Signed Values Consider the function:

$$((X1 \text{ AND } (\text{NOT } X2)) \text{ OR } (X3 \text{ AND } (\text{NOT } X4)))$$

Which has partition selection coefficients -6, 6, -6, 6. If partition selection were based on signed value then the following matrices would be tried.

X1 & X3									
X2	0	0	0	0					
&	1	0	1	0					
X4	1	1	0	0					
	1	1	1	0					
X1 & X2 & X3					X1 & X2 & X4				
X4	0	0	0	0	1	0	1	0	
	1	1	0	0	1	1	1	0	
X1 & X3 & X4					X2 & X3 & X4				
X2	0	0	1	0	1	1	1	1	
	0	0	1	0	0	0	1	0	
					X3	0	0	0	0
						0	1	0	1
						1	1	1	0
					X1	0	0	1	1
						0	0	0	0
						1	0	1	0

These would not be tried.

X1 & X2					X1 & X4				
X3	0	0	1	0	X2	0	0	1	1
&	0	0	1	0	&	1	0	1	1
X4	1	1	1	1	X3	0	0	0	0
	0	0	1	0		1	0	1	0

If signed values were used instead of absolute values the optimal decomposition would have been missed. Use of absolute values of the coefficients, however, increases the search space since more coefficients of the same absolute value are likely to exist. In this rather simple example all possible partition matrices would have been tried. Sign changes can result from negation of a variable or a function (i.e., NOT(OR) => NAND).

2.2.4 A Final Example

An attempt was made to cause the partition selection algorithm to fail by "designing" a function which would force an incorrect grouping. This was done by combining the same function on different sets of input variables with an OR gate. Since each subfunction would have the same coefficients, and simply combining them with an OR gate would cause this to be transferred to the output causing an incorrect grouping.

Choose an arbitrary function:

$$G(x, y, z) = (x \text{ AND } (y \text{ OR } z))$$

Combine two of these with an OR gate:

$$F(X_1, \dots, X_6) = G(X_1, X_2, X_3) \text{ OR } G(X_4, X_5, X_6)$$

The coefficients generated for this function are 30, 10, 10, 30, 10 and 10 respectively. This suggests grouping X_1 and X_4 ; and X_2 , X_3 , X_5 , and X_6 . This appears to be an the desired incorrect grouping, since the intuitively correct groupings would have been X_1 , X_2 and X_3 ; and X_4 , X_5 and X_6 .

However, if the partition is formed:

$X_2, X_3, X_5 \text{ \& } X_6$

X_1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
&	0	1	1	1	0	1	1	1	0	1	1	1	0	1	1	1
X_4	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1
	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
	0	1	1	1	0	1	1	1	0	1	1	1	0	1	1	1
	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1

=> X_7
=> X_8

Further decompositions result.

F					X7					X8				
$X_1 \text{ \& } X_7$					$X_2 \text{ \& } X_3$					$X_2 \text{ \& } X_3$				
X_4	0	0	0	1	X_5	0	1	1	1	X_5	0	0	0	0
&	0	0	0	1	&	0	1	1	1	&	1	1	1	1
X_8	0	0	0	1	X_6	0	1	1	1	X_6	1	1	1	1
	1	1	1	1		0	1	1	1		1	1	1	1

Notice that X_7 is vacuous in X_5 and X_6 ; X_8 is vacuous in X_2 and X_3 . In fact:

$$\begin{aligned} X_7 &= X_2 \text{ OR } X_3, \\ X_8 &= X_5 \text{ OR } X_6, \end{aligned}$$

and

$$F = (X_1 \text{ AND } X_7) \text{ OR } (X_4 \text{ AND } X_8)$$

which was the original function.

Several more experiments like this were performed using AND and XOR gates and more variables with similar results. An initial apparently incorrect grouping of subfunctions decomposed to the original function.

3.0 Summary

The procedure for calculating coefficients has been implemented in a BASIC program. The procedure for selecting partitions has not yet been automated. Because of the way the algorithm selects coefficients at various levels of decomposition, it needs to be implemented in a program which completes the decomposition. Examining initial groupings does not mean improperly selected groupings will not be eliminated at a later stage of decomposition via generation of vacuous variables (See example 2.2.4).

The partition selection algorithm described seems to be a good method for reducing the search space for functional decomposition programs like AFD. There is no theoretical basis for this method; this needs development. It is yet not known if it covers the entire search space needed for optimal decompositions, since at this time only a few hand done, machine assisted, examples have been tried. Even if it does not provide complete coverage of the search space, it usually provides a low cost decomposition and could be a good starting point for limiting the search tree via pruning techniques.

There are still a number of questions about this technique. Does it cover the entire required search space? In what order should grouping be tried? How well does it work with partial functions? Only a few examples have been tried, but the technique seems promising. Further experimentation is needed. Incorporating this algorithm in the Function Learning Synthesis HotBench the C++ version of AFD is required to answer these and other questions.

Appendix B.

Experimental Data

Summary of data: The following pages give a summary of the experimental runs for actual data.

Key:

Baseline / Structured / Structured P[0] compliment / Structured odd child
fix - refers to the particular algorithm used.

all functions - more than one partition selection leads to an optimal decomposition
functions with one decomposition - only one partition selection leads to an optimal decomposition

Negative decomposition suppressed / Negative decomposition allowed - if values greater than table size were allowed in the fitness factor.

DFC Found / DFC not Found - number of functions in which the algorithm found or did not find the optimal partition.

faster / slower - number of cases the algorithm ran slower or faster than a random search.
chance / actual - expected number of partitions searched randomly before an optimal partition is found and number of average number of partitions actually searched by the genetic algorithm before it terminated.

hits - number of optimal partitions in the function.

improvement - the actual improvement factor over chance (summed over runs - not average)

		Baseline		all functions			
		Negative decomposition suppressed					
		DFC not Found					
DFC Found faster		faster	slower	chance	actual	hits	improvement
18	24		8	3	3.43	21	0.74
34	16			5.72	5.53	3	1.14
28	22			3.9	4.1	12	0.87
29	21			5.72	5.86	3	0.91
47	3			7	5.3	1	3.25
38	12			5.72	5.39	3	1.26
41	9			5.35	4.52	4	1.78
38	12			5	4.5	5	1.41
19	31			3.1	4	20	0.54
26	24			2.58	4.44	15	0.28
50	0			7	5.39	1	3.05
44	6			7	6.27	1	1.66
22	28			7	7.14	1	0.91
32	18			4.39	4.49	8	0.93
37	13			5.3	4.8	4	1.41
22	28			4.58	5.31	7	0.60
38	12			1	1.5	126	0.71
41	9			6.22	5.34	2	1.84
38	12			6.23	5.79	2	1.36
50	0			7	5.38	1	3.07
50	0			7	5.14	1	3.63
50	0			7	5.3	1	3.25
41	9			7	6.26	1	1.67
40	10			7	6.26	1	1.67
40	9		1	7	6.23	1	1.71
21	26		3	7	7.22	1	0.86
934	354		12	1300			40.50
							1.56 times better than chance
71.85%	27.23%	0.00%	0.92%				

		Baseline		all functions			
		Negative decomposition allowed					
		DFC not Found					
DFC Found faster	slower	faster	slower	chance	actual	hits	improvement
17	33			3	3.49	21	0.71
30	20			5.72	5.6	3	1.09
24	26			3.9	4.42	12	0.70
36	14			5.72	5.4	3	1.25
49	1			7	5.89	1	2.16
38	12			5.72	5.2	3	1.43
33	17			5.35	5.1	4	1.19
26	24			5	5.24	5	0.85
5	45			3.1	4.36	20	0.42
17	33			3.5	3.15	15	1.27
49	1			7	5.89	1	2.16
49	1			7	5.89	1	2.16
20	30			7	7.11	1	0.93
30	20			4.39	4.85	8	0.73
31	19			5.3	5.39	4	0.94
21	29			4.58	5.11	7	0.69
36	14			1	1.77	126	0.59
44	6			6.22	5.43	2	1.73
37	13			6.23	5.76	2	1.39
50	0			7	5.33	1	3.18
49	1			7	5.58	1	2.68
50	0			7	5.39	1	3.05
47	3			7	6.13	1	1.83
50	0			7	5.17	1	3.56
38	12			7	6.57	1	1.35
30	20			7	6.9	1	1.07
906	394		0	1300			39.08
							1.50 times better than chance
69.69%	30.31%	0.00%	0.00%				

		Structured		all functions			
		Negative decomposition suppressed					
DFC Found		DFC not Found		chance	actual	hits	improvement
faster	slower	faster	slower				
131	119			3	4.36	21	0.39
105	52	93		5.72	3.11	3	6.11
205	1	44		3.9	2.9	12	2.00
48	0	202		5.72	3.3	3	5.35
153	0	97		7	3.73	1	9.65
77	173			5.72	3.28	3	5.43
216		34		5.3	3.48	4	3.53
115	2	132	1	5	3.07	5	3.81
133	117			3.1	3.62	20	0.70
128	122			3.58	3.99	15	0.75
250	0	0	0	7	4.27	1	6.63
13	0	237		7	4.38	1	6.15
19	0	231		7	4.44	1	5.90
159	6	78	7	4.39	3.38	8	2.01
131	1	114	4	5.3	3.81	4	2.81
123	127			4.58	5.03	7	0.73
186	64			1	0.8	126	1.15
78	0	172		6.22	4.26	2	3.89
156	18	76		6.23	5.06	2	2.25
250	0			7	4.63	1	5.17
214	0	36		7	3.79	1	9.25
210	0	40		7	3.72	1	9.71
137	0	113		7	4.3	1	6.50
216	0	34		7	3.74	1	9.58
106	0	144		7	4.98	1	4.06
31	0	216	3	7	5.56	1	2.71
3590	802	2093	15	6500			116.22
percent							4.47
55.23%	12.34%	32.20%	0.23%				times better than chance

		Structured		all functions				
		Negative decomposition allowed						
		DFC not Found						
DFC Found faster	slower	faster	slower	chance	actual	hits	improvement	
118	101	31			3	4.26	21	0.42
107	33	110			5.72	4.9	3	1.77
154	1	95			3.9	2.6	12	2.46
47	0	202	1		5.72	2.8	3	7.57
127	0	123			7	3.3	1	13.00
51	0	199			5.72	2.55	3	9.00
112	138	0			5.3	2.6	4	6.50
41	8	188	13		5	3.08	5	3.78
72	53	125			3.1	3.4	20	0.81
70	56	123	1		3.58	3.69	15	0.93
250	0	0	0		7	4.25	1	6.73
98	0	152			7	3.35	1	12.55
2	0	248			7	3.34	1	12.64
68	6	175	1		4.39	2.06	8	5.03
76	8	166	0		5.3	2.37	4	7.62
26	23	199	2		4.58	3.24	7	2.53
175	0	75			1	0.95	126	1.04
67	0	183	0		6.22	3.9	2	4.99
127	15	108			6.23	4	2	4.69
250	0				7	4.23	1	6.82
64	0	186			7	2.63	1	20.68
184	0	66			7	3.55	1	10.93
184	0	66			7	3.51	1	11.24
170	0	80			7	3.4	1	12.13
41	0	209			7	3.24	1	13.55
11	1	234	4		7	4.37	1	6.19
2692	443	3343	22	6500			185.58	
percent							7.14	times better than chance
41.42%	6.82%	51.43%	0.34%					

		Structured P[0] compliment Negative decomposition allowed		all functions			
DFC Found faster	slower	DFC not Found faster	slower	chance	actual	hits	improvement
104	146	0		3	3.72	21	0.61
154	63	33		5.72	5.3	3	1.34
227	1	22		3.9	3	12	1.87
55	0	195		5.72	3.5	3	4.66
213	0	37		7	3.8	1	9.19
81	0	169		5.72	3.36	3	5.13
219	0	31		5.3	3.55	4	3.36
116	0	128	6	5	3.69	5	2.48
115	3	131	1	3.1	3.76	20	0.63
107	0	140	3	3.58	3.72	15	0.91
250	0	0	0	7	4.29	1	6.54
119	0	131		7	3.77	1	9.38
13	0	237		7	4.25	1	6.73
192	6	49	3	4.39	3.26	8	2.19
148	0	100	2	5.3	3.8	4	2.83
121	0	129	0	4.58	4.97	7	0.76
180	0	70		1	0.96	126	1.03
197	0	53	0	6.22	4.14	2	4.23
230	15	5		6.23	5.5	2	1.66
250	0			7	4.37	1	6.19
250	0	0		7	3.7	1	9.85
250	0	0		7	3.72	1	9.71
139	0	111		7	4.31	1	6.45
250	0	0		7	3.6	1	10.56
119	0	130	1	7	5.1	1	3.73
48	1	195	6	7	5.7	1	2.46
4147	235	2096	22	6500			114.48
percent							4.40
63.80%	3.62%	32.25%	0.34%				times better than chance

		Structured odd child fix Negative decomposition allowed		all functions				
DFC Found faster	slower	DFC not Found faster	slower	chance	actual	hits	improvement	
113	137	0		3	4.5	21	0.35	
153	69	28		5.72	5.29	3	1.35	
216	0	33	1	3.9	3.1	12	1.74	
59	0	191		5.72	3.45	3	4.82	
202	0	48		7	3.88	1	8.69	
82	0	168		5.72	3.28	3	5.43	
226	0	24		5.3	3.54	4	3.39	
93	1	151	5	5	3.84	5	2.23	
127	0	123	0	3.1	3.68	20	0.67	
132	118	0	0	3.58	3.9	15	0.80	
250	0	0	0	7	4.29	1	6.54	
130	0	120		7	3.7	1	9.85	
20	0	230		7	4.49	1	5.70	
178	8	59	5	4.39	3.36	8	2.04	
144	0	104	2	5.3	3.87	4	2.69	
134	116	0	0	4.58	4.92	7	0.79	
192	0	58		1	0.84	126	1.12	
190	0	60	0	6.22	4.31	2	3.76	
230	15	0	5	6.23	5.45	2	1.72	
250	0			7	4.27	1	6.63	
250	0	0		7	3.8	1	9.19	
250	0	0		7	3.7	1	9.85	
134	0	116		7	4.34	1	6.32	
250	0	0		7	3.73	1	9.65	
115		134	1	7	5.1	1	3.73	
50	1	195	4	7	5.7	1	2.46	
4170	465	1842	23	6500			111.52	
percent							4.29	times better than chance
64.15%	7.15%	28.34%	0.35%					

		functions with one decomposition		chance	actual	hits	improvement	
DFC Found faster	slower	DFC not Found faster	slower					
47	3			7	5.3	1	3.25	
50	0			7	5.39	1	3.05	
44	6			7	6.27	1	1.66	
22	28			7	7.14	1	0.91	
50	0			7	5.38	1	3.07	
50	0			7	5.14	1	3.63	
50	0			7	5.3	1	3.25	
41	9			7	6.26	1	1.67	
40	10			7	6.26	1	1.67	
40	9		1	7	6.23	1	1.71	
21	26		3	7	7.22	1	0.86	
455	91		4	550			24.72	
82.73%	16.55%	0.00%	0.73%				2.25	times better than chance

DFC Found faster	slower	functions with one decomposition DFC not Found faster	slower	chance	actual	hits	improvement
49	1				7	5.89	1
49	1				7	5.89	1
49	1				7	5.89	1
20	30				7	7.11	1
50	0				7	5.33	1
49	1				7	5.58	1
50	0				7	5.39	1
47	3				7	6.13	1
50	0				7	5.17	1
38	12				7	6.57	1
30	20				7	6.9	1
481	69	0	0	550			24.11
87.45%	12.55%	0.00%	0.00%				2.19 times better than chance

		functions with one decomposition				chance	actual	hits	improvement
DFC Found faster	slower	DFC not Found faster	slower						
153	0	97				7	3.73	1	9.65
250	0	0	0			7	4.27	1	6.63
13	0	237				7	4.38	1	6.15
19	0	231				7	4.44	1	5.90
250	0					7	4.63	1	5.17
214	0	36				7	3.79	1	9.25
210	0	40				7	3.72	1	9.71
137	0	113				7	4.3	1	6.50
216	0	34				7	3.74	1	9.58
106	0	144				7	4.98	1	4.06
31	0	216	3			7	5.56	1	2.71
1599	0	1148	3	2750					75.31
58.15%	0.00%	41.75%	0.11%						6.85 times better than chance

		functions with one decomposition				chance	actual	hits	improvement
DFC Found faster	slower	DFC not Found faster	slower						
127	123	0			7	3.3	1	13.00	
250	0	0	0		7	4.25	1	6.73	
98	0	152			7	3.35	1	12.55	
19	0	231			7	3.34	1	12.64	
250	0				7	4.23	1	6.82	
64	0	186			7	2.63	1	20.68	
184	0	66			7	3.55	1	10.93	
184	0	66			7	3.51	1	11.24	
170	0	80			7	3.4	1	12.13	
41	0	209			7	3.24	1	13.55	
11	1	234	4		7	4.37	1	6.19	
1398	124	1224	4	2750				126.44	
50.84%	4.51%	44.51%	0.15%					11.49	times better than chance

		functions with one decomposition				chance	actual	hits	improvement
DFC Found faster	slower	DFC not Found faster	slower						
213	0	37				7	3.8	1	9.19
250	0	0	0			7	4.29	1	6.54
119	0	131				7	3.77	1	9.38
13	0	237				7	4.25	1	6.73
250	0					7	4.37	1	6.19
250	0	0				7	3.7	1	9.85
250	0	0				7	3.72	1	9.71
139	0	111				7	4.31	1	6.45
250	0	0				7	3.6	1	10.56
119	0	130	1			7	5.1	1	3.73
48	1	195	6			7	5.7	1	2.46
1901	1	841	7		2750				80.80
69.13%	0.04%	30.58%	0.25%						7.35 times better than chance

		functions with one decomposition				chance	actual	hits	improvement
DFC Found faster	slower	DFC not Found faster	slower						
202	0	48			7	3.88	1	8.69	
250	0	0	0		7	4.29	1	6.54	
130	0	120			7	3.7	1	9.85	
20	0	230			7	4.49	1	5.70	
250	0				7	4.27	1	6.63	
250	0	0			7	3.8	1	9.19	
250	0	0			7	3.7	1	9.85	
134	0	116			7	4.34	1	6.32	
250	0	0			7	3.73	1	9.65	
115		134	1		7	5.1	1	3.73	
50	1	195	4		7	5.7	1	2.46	
1901	1	843	5	2750				78.62	
69.13%	0.04%	30.65%	0.18%					7.15	times better than chance

Summary of data: The following pages give a summary of the experimental runs for simulated data.

Key:

baseline / Structured - refers to the particular algorithm used.

groups - the number of groups inputs are divided into for row column selection

hits - number of optimal partitions in the function.

improvement - the actual improvement factor over chance (summed over runs - not average)

baseline

groups	2			
hits	1	1	1	1
variables	6	8	10	12
chance	5	7	9	11
actual	4.13	5.54	6.5	7.73
improvement	0.87	1.46	2.5	3.27
hits	2	2	2	2
variables	6	8	10	12
chance	4.75	6.75	8.75	10.75
actual	3.86	5.37	6.85	7.78
improvement	0.89	1.38	1.9	2.97
groups	3			
hits	1	1		1
variables	6	9		12
chance	5	8		11
actual	3.9	5.3		6.66
improvement	1.1	2.7		4.34
hits	3	3		3
variables	6	9		12
chance	4.5	7.5		10.5
actual	3.29	5.23		6.49
improvement	1.21	2.27		4.01
hits	6	6		6
variables	6	9		12
chance	3.75	6.75		9.75
actual	2.42	4.77		6
improvement	1.33	1.98		3.75
groups	4			
hits	1			1
variables	12			12
chance	11			11
actual	5			6.2
improvement	6			4.8
hits	3			3
variables	12			12
chance	10.5			10.5
actual	4.5			6.06
improvement	6			4.44
hits	7			7
variables	12			12
chance	9.5			9.5
actual	3.9			5.8
improvement	5.6			3.7
hits	14			14
variables	12			12
chance	7.75			7.75
actual	3.2			5.4
improvement	4.55			2.35

structured

groups	2			
hits	1	1	1	1
variables	6	8	10	12
chance	5	7	9	11
actual	3.56	5.14	6.29	7.44
improvement	1.44	1.86	2.71	3.56
hits	2	2	2	2
variables	6	8	10	12
chance	4.75	6.75	8.75	10.75
actual	3.24	4.57	5.67	6.81
improvement	1.51	2.18	3.08	3.94
groups	3			
hits	1	1		1
variables	6	9		12
chance	5	8		11
actual	3.59	4.73		6.39
improvement	1.41	3.27		4.61
hits	3	3		3
variables	6	9		12
chance	4.5	7.5		10.5
actual	2.92	4.5		6.06
improvement	1.58	3		4.44
hits	6	6		6
variables	6	9		12
chance	3.75	6.75		9.75
actual	2.25	3.75		5.28
improvement	1.5	3		4.47
groups	4			
hits	1			1
variables	12			12
chance	11			11
actual	4.27			6.1
improvement	6.73			4.9
hits	3			3
variables	12			12
chance	10.5			10.5
actual	3.77			5.96
improvement	6.73			4.54
hits	7			7
variables	12			12
chance	9.5			9.5
actual	3.15			5.21
improvement	6.35			4.29
hits	14			14
variables	12			12
chance	7.75			7.75
actual	2.5			4.4
improvement	5.25			3.35